DIRECTION DE LA RECHERCHE TECHNOLOGIQUE

CEA/SACLAY

DEPARTEMENT DES TECHNOLOGIES
DES SYSTEMES INTELLIGENTS

SERVICE OUTILS LOGICIELS

Service Outils Logiciels

Fluctuat C - User manual

Static analysis of the precision of floating point computations.

par

E. Goubault, S. Putot, K. Tekkal et F. Vedrine
DTSI/SOL/MEASI (CEA)

DIRECTION DE LA RECHERCHE TECHNOLOGIQUE
LABORATOIRE D'INTEGRATION DES SYSTEMES ET DES TECHNOLOGIES
DEPARTEMENT DES TECHNOLOGIES DES SYSTEMES INTELLIGENTS
SERVICE OUTILS LOGICIELS
CEA/SACLAY - 91191 GIF-SUR-YVETTE CEDEX
TEL 01 69 08 33 53 - FAX 01 69 08 83 95 - E-MAIL jan.stransky@cea.fr
Etablissement public à caractère industriel et commercial
R.C.S PARIS B 775 685 019

Table des matières

1 Introduction and principle

The manipulation of real numbers by computers is approximated by floating-point arithmetic, which uses a finite representation of numbers. This implies that a (small in general) rounding error may be committed at each operation. Although this approximation is accurate enough for most applications, there are some cases where results become irrelevant because of the precision lost at some stages of the computation, even when the underlying numerical scheme is stable.

We present a tool for studying the propagation of rounding errors in floating-point computations. Its aim is to detect automatically a possible catastrophic loss of precision, and its source. The tool is intended to cope with real industrial problems, and we believe it is specially appropriate for critical instrumentation software. On these numerically quite simple programs, we believe our tool will bring some very helpful information, and allow us to find possible programming errors such as potentially dangerous double/float conversions, or blatant instabilities or losses of accuracy. The techniques used being those of static analysis, the tool will not compete on numerically intensive codes with a numerician's study of stability. Neither is it designed for helping find better numerical schemes. But, it is automatic and in comparison with a study of sensitivity to data, brings about the contribution of rounding errors occurring at every intermediary step of the computation. Moreover, static analyses are sure (but may be pessimistic) and consider a set of possible executions and not just one, which is the essential requirement a verification tool for critical software must meet.

The prototype delivers, for an ANSI C program, and for each scalar variable, and at the last control point of the function which is analysed :

– an interval, superset of possible floating-point values,
– an imprecision error, with respect to the semantics involving real numbers, decomposed by contribution of each line of **C** code,
– an interval, superset of the higher-order imprecision (global) error with respect to the "real" semantics.

In this section, we briefly introduce the semantics of floating-point numbers with errors which is detailed in ref. [**?**, **?**]. Let us examine a simple calculation involving two values $a_{\mathbb{F}} = 621.3$ and $b_{\mathbb{F}} = 1.287$. For the sake of simplicity, we assume that $a_{\mathbb{F}}$ and $b_{\mathbb{F}}$ belong to a simplified set of floating-point numbers composed of a mantissa of four digits written in base 10. We assume that initial errors are attached to $a_{\mathbb{F}}$ and $b_{\mathbb{F}}$, and we write $a = 621.3 + 0.05\vec{\varphi}_1$ and $b = 1.287 + 0.0005\vec{\varphi}_2$ to indicate that the value of the initial error on $a_{\mathbb{F}}$ (resp. $b_{\mathbb{F}}$) is 0.05 (resp. 0.0005). Formal variables $\vec{\varphi}_1$ and $\vec{\varphi}_2$ are related to the static control points at which these errors were introduced (either because there is an imprecision on an input, or because of an operation). a and b are called *floating-point numbers with errors*, and represent the result of the same computation as $a_{\mathbb{F}}$ and $b_{\mathbb{F}}$ but using real numbers instead of floating-point numbers.

Let us focus on the product $a_{\mathbb{F}} \times b_{\mathbb{F}}$ whose exact result is $a_{\mathbb{F}} \times b_{\mathbb{F}} = 799.6131$. This calculation carried out with floating-point numbers with errors yields $a \times b = 799.6131 + 0.06435\vec{\varphi}_1 + 0.31065\vec{\varphi}_2 + 0.000025\vec{\varphi}_1\vec{\varphi}_2$ which we rewrite as $a \times b = 799.6131 + 0.06435\vec{\varphi}_1 + 0.31065\vec{\varphi}_2 + 0.000025\vec{\varphi}_{hi}$. The rewriting step is made to keep only one formal variable per coefficient and obeys the following rule. The indices of the formal variables $\vec{\varphi}_1$, $\vec{\varphi}_2$ etc. are viewed as words of the alphabet of the control points and the product of two variables yields a new formal variable by concatenation of the index words. A word of length one describes a first-order error and the special word hi is used to identify all the words of length greater than one.

The difference between $a_{\mathbb{F}} \times b_{\mathbb{F}}$ and 621.35×1.2875 is 0.375025 and this error stems from the fact that the initial error on a (resp. b) was multiplied by b (resp. a) and that a second-order error corresponding to the multiplication of both errors was introduced. So, at the end of the calculation, the contribution to the global error of the initial error on a (resp. b) is 0.06435 (resp. 0.31065) and corresponds to the coefficient attached to the formal variable $\vec{\varphi}_1$ (resp. $\vec{\varphi}_2$). Finally, the number 799.6131 has too many digits to be representable in our floating-point number system and we refer to the IEEE-754 norm for floating-point arithmetic to determine how the values are rounded [**?**]. Let $\mathbb{F}$ be either the set of simple or double precision floating-point numbers. The norm fully specifies the function $\uparrow_\circ : \mathbb{R} \to \mathbb{F}$ which returns the rounded value of a real number $r \in \mathbb{R}$ with respect to the current rounding mode $\circ$ [**?**, **?**]. In addition, it ensures that the elementary operations are correctly rounded, i.e. for any operator $\Diamond \in \{+,-,\times,\div\}$, we have :

$$\forall f_1, f_2 \in \mathbb{F},\ f_1 \Diamond_{\mathbb{F}} f_2 = \uparrow_\circ (f_1 \Diamond_{\mathbb{R}} f_2) \tag{1}$$

The function $\downarrow_\circ : \mathbb{R} \to \mathbb{F}$ that returns the roundoff error is then defined by

$$\forall x \in \mathbb{R},\ \downarrow_\circ (x) = x - \uparrow_\circ (x)\ .$$

In Equation (1), $\Diamond_{\mathbb{F}}$ and $\Diamond_{\mathbb{R}}$ denote the same operation over $\mathbb{F}$ and $\mathbb{R}$. Assuming that the machine we are using conforms to that standard, we may claim that the computed floating-point number for our example is $\uparrow_\circ (799.6131) = 799.6$ and that a new first error term $0.0131\vec{\varphi}_\times$ is introduced by the multiplication. To sum up, we have

$$a \times b = 799.6 + 0.06435\vec{\varphi}_1 + 0.31065\vec{\varphi}_2 + 0.000025\vec{\varphi}_{hi} + 0.0131\vec{\varphi}_\times \tag{2}$$

At first sight, one would think that the precision of the calculation mainly depends on the initial error on a since it is 100 times larger than the one on b. However the above result indicates that the final error is mainly due to the initial error on b. Hence, to improve the precision of the final result, one should first try to increase the precision on b (whenever possible).

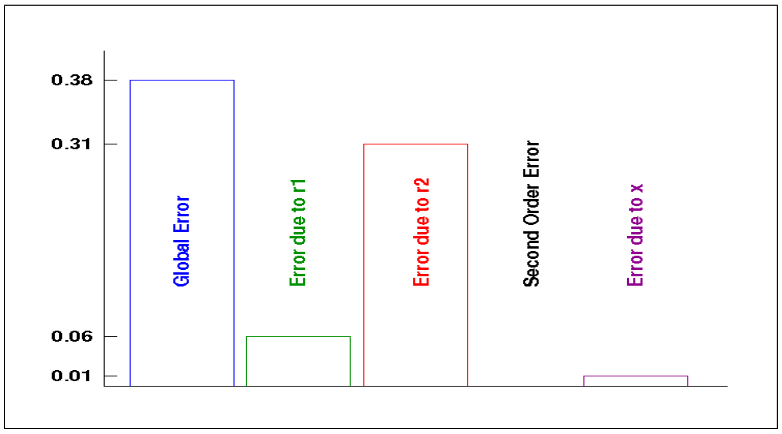


FIGURE 1 – Graphical representation of the errors defined in Equation (2).

The errors arising during the calculation of $a \times b$, which are given in Equation (2), can be represented by a graph which x-axis enumerates the control points of the program and which y-axis indicates the magnitude of the errors related to each point. The graph corresponding to Equation (2) is given in Figure 1. In general, Fluctuat displays such a graph for each variable visible at the end of the program being analysed (see Figure 4 in Section 2.1).

More generally, the difference between the result x of a computation in real numbers, and the result f^x of the same computation using floating-point numbers, is expressed as

$$x = f^x + \sum_{\ell \in L \cup \{hi\}} \omega^x_\ell \vec{\varphi}_\ell\ . \tag{3}$$

In this relation, a term $\omega^x_\ell \vec{\varphi}_\ell$, $\ell \in L$ denotes the contribution to the global error of the first-order error introduced by the operation labelled ℓ. The value of the error $\omega^x_\ell \in \mathbb{R}$ expresses the rounding error committed at label ℓ, and its propagation during further computations on variable x. Variable $\vec{\varphi}_\ell$ is a formal variable, associated to point ℓ, and with value 1. Errors of order higher than one, coming from non-affine operations, are grouped in one term associated to special label hi, and we note $\mathcal{L} = L \cup \{hi\}$.

The result of an arithmetic operation $\Diamond$ at point ℓ_i contains the combination of existing errors on the operands, plus a new roundoff error term $\downarrow_\circ (f^x \Diamond f^y)\vec{\varphi_\ell}$. For addition and subtraction, the errors are added or subtracted componentwise :

$$x +^{\ell_i} y = \uparrow_\circ (f^x + f^y) + \sum_{\ell \in \mathcal{L}} (\omega_\ell^x + \omega_\ell^y)\vec{\varphi_\ell} + \downarrow_\circ (f^x + f^y)\vec{\varphi_{\ell_i}} \, .$$

The multiplication introduces higher order errors, we write :

$$x \times^{\ell_i} y = \uparrow_\circ (f^x f^y) + \sum_{\ell \in \mathcal{L}} (f^x \omega_\ell^y + f^y \omega_\ell^x)\, \vec{\varphi_\ell} + \sum_{\ell_1 \in \mathcal{L},\ \ell_2 \in \mathcal{L}} \omega_{\ell_1}^x \omega_{\ell_2}^y \, \vec{\varphi_{hi}} + \downarrow_\circ (f^x f^y)\vec{\varphi_{\ell_i}} \, .$$

Interpretation of other operation were also defined, that we do not detail here.

A natural abstraction of the coefficients in expression (3), is obtained using intervals. The machine number f^x is abstracted by an interval of floating-point numbers, each bound rounded to the nearest value in the type of variable x. The error terms $\omega_i^x \in \mathbb{R}$ are abstracted by intervals of higher-precision numbers, with outward rounding. This is implemented in the default analysis mode of Fluctuat, called "non relational abstract domain" at the graphic interface.

However, results with this abstraction suffer from the over-estimation problem of interval methods. If the arguments of an operation are correlated, the interval computed with interval arithmetic may be significantly wider than the actual range of the result. An implicitly relational version of this domain [**?**, **?**] has been added more recently. Relying on affine arithmetic instead of interval arithmetic to compute the value f^x, it is in general a little more costly in memory and computation time, but more accurate. It is briefly described in subsection 1.1, and results with this analysis are shown in section 4. We also plan to add a mode that implements a relational domain for the errors.

Finally, an extension of the non-relational analysis has been added to analyse the amplitude and provenance of errors coming from a fixed-point implementation of a program. This implementation is given for the time being by assertions on the fixed-point format (word length, length of integer and fractional part) used for each float or double variable. A first prototype exists, of a tool that allows to determine automatically, from a floating-point program, fixed-point formats that would give the same order of accuracy for outputs of the program with the fixed-point semantics. This tool relies on analyses with Fluctuat with floating-point and fixed-point semantics, and on heuristics to progressively improve the fixed-point format.

1.1 Implicitly relational domain for the value f^x

Affine arithmetic was proposed by Comba, De Figueiredo and Stolfi [**?**, **?**], as a solution to the overestimation in interval arithmetic, for bounding the result of a computation in real numbers. After a short introduction on affine arithmetic, we describe very shortly the domain we proposed in [**?**, **?**] and implemented in Fluctuat, to estimate the floating-point value f^x in expression (3), domain which is based on affine arithmetic.

1.1.1 Affine arithmetic on real numbers

In affine arithmetic, a quantity x is represented by an affine form, which is a polynomial of degree one in a set of noise terms ε_i :

$$\hat{x} = \alpha_0^x + \alpha_1^x \varepsilon_1 + \ldots + \alpha_n^x \varepsilon_n, \text{ with } \varepsilon_i \in [-1, 1] \text{ and } \alpha_i^x \in \mathbb{R}. \tag{4}$$

Each noise symbol ε_i stands for an independent component of the total uncertainty on the quantity x, its value is unknown but bounded in [-1,1] ; the corresponding coefficient α_i^x is a known real value, which gives the magnitude of that component. The same noise symbol can be shared by several quantities, indicating correlations among them. These noise symbols can be used not only for modelling uncertainty in data or parameters, but also uncertainty coming from computation.

The assignment of a variable x whose value is given in a range $[a, b]$, introduces a noise symbol ε_i :

$$\hat{x} = (a + b)/2 + (b - a)/2\,\varepsilon_i.$$

The result of linear operations on affine forms, applying polynomial arithmetic, can easily be interpreted as an affine form. For example, for two affine forms $\hat{x}$ and $\hat{y}$, and a real number r, we get

$$\begin{array}{rcl} \hat{x} + \hat{y} & = & (\alpha_0^x + \alpha_0^y) + (\alpha_1^x + \alpha_1^y)\varepsilon_1 + \ldots + (\alpha_n^x + \alpha_n^y)\varepsilon_n \\ \hat{x} + r & = & (\alpha_0^x + r) + \alpha_1^x\varepsilon_1 + \ldots + \alpha_n^x\varepsilon_n \\ r\hat{x} & = & r\alpha_0^x + r\alpha_1^x\varepsilon_1 + \ldots + r\alpha_n^x\varepsilon_n \end{array}$$

For non affine operations, the result applying polynomial arithmetic is not an affine form : we select an approximate linear resulting form, and bounds for the approximation error committed using this approximate form are computed, that create a new noise term added to the linear form.

1.1.2 Computation of the floating-point value

Using affine arithmetic for the estimation of floating-point values needs some adaptation. Indeed, the correlations that are true on real numbers after an arithmetic operation, are not exactly true on floating-point numbers. Consider for example two independent variables x and y that both take their value in the interval [0,2], and the arithmetic expression $((x + y) - y) - x$. Using affine arithmetic in the classical way, we write $x = 1 + \varepsilon_1$, $y = 1 + \varepsilon_2$, and we get zero as result of the expression. This is the expected result, provided this expression is computed in real numbers. But if we take x as the nearest floating-point value to 0.1, and $y = 2$, then the floating-point result is $-9.685755e - 8$.

In order to model the floating-point computation, a rounding error must thus be added to the affine form resulting from each arithmetic operation. We decompose the floating-point value f^x of a variable x resulting from a trace of operations, in the real value of this trace of operations r^x, plus the sum of errors δ^x accumulated along the computation, $f^x = r^x + \delta^x$, in which r^x is computed using affine arithmetic.

A natural idea for the error δ^x is to add an interval term, or a new noise term for each operation. However, this is not always satisfying. We thus use an error expressed as the combination of a maximum error (interval term or noise symbol), and the errors associated to the minimum and maximum real values. These errors on the bounds can often be computed more accurately and are the only information needed to compute the bounds of the floating-point value. But the maximum error may still be needed for the computation of further arithmetic operations.

For static analysis, we need to define an order, and meet and join operations over this domain. This is not obvious because there are no natural union and intersection on affine forms. Also, affine arithmetic was defined with real coefficients : in order to have a computable abstraction, we have to consider the errors due to the use of (arbitrary precision) floating-point numbers instead of real numbers. We refer the user to [**?**, **?**] for more precision about these points, but we are still working to improve the first solutions proposed in these references.

2 Using Fluctuat

2.1 Introduction : the phases of analysis

There are different phases in the use of the tool. For a given problem the user has to go through three phases : project creation, static analysis and visualisation of the results.

Project creation : A project is constituted of one or several sources files (**.c**) given through the user interface (see figure 2). A directory containing included files can be specified, as well as some C compiling options (essentially the -D option).
Then the pre-analyser precompiles the ANSI C program constructed to see if everything is correct, and preprocesses it. It pre-computes the arguments, types etc. of each function. There are a number of intermediary files generated by this phase, that should not be edited (the **project.xml** in particular).
This phase is launched either by choosing **Create Project** in the **File** menu, or by clicking on the first button from the left on the top of the main window of the interface.
This has only to be done once, and for the next phase (the analysis phase), you can directly give the name of the project to the analyser (in general a **project.xml** file) by selecting "Open Project".

Create project button

Open project button

Static analysis : The parameters that tune the precision of the static analyser, can be edited when you ask for the analysis, via the dialog window shown in Figure 3. This dialog box is automatically displayed when the project creation step is completed, or when **Start Analysis** (or second button from the left on the top of the interface) is selected.
When, still in the **Analysis** menu, **Quick Analysis** is selected (or third button from the left on the top of the interface), the parameters used are those defined for the previous analysis.
The values of the parameters for the current analysis are displayed when selecting **View Parameters** in the **Analysis** menu (or clicking on the fourth button on the top of the interface).

Start analysis button

Quick analysis button

View parameters button

How these parameters need to be tuned is discussed in Section 2.4.

Visualisation of the results : At the end of the analysis, the main window of the analyser displays the results of the analysis, and more information can be accessed from this window (see subsection 2.5).
In case a preprocessing script is selected, the program printed in the left-hand window is the result of this script applied to the given sources.

Each of theses phases are detailed in the following sections.

2.2 Project creation

When selecting **Create Project**, the window shown in figure 2 opens.

2.2.1 General information

This section is automaticly filled when you select a source file. You can choose the name and the used directory of the project. The default behavior is to complete with the name and the directory of the first selected file.

- first, a number of source files can be selected using the **Add File** button,
- then, directories containing the header files .h can be selected through the **Add dir** button,
- in the case when several source files were selected, a preprocessing script must be specified, default is

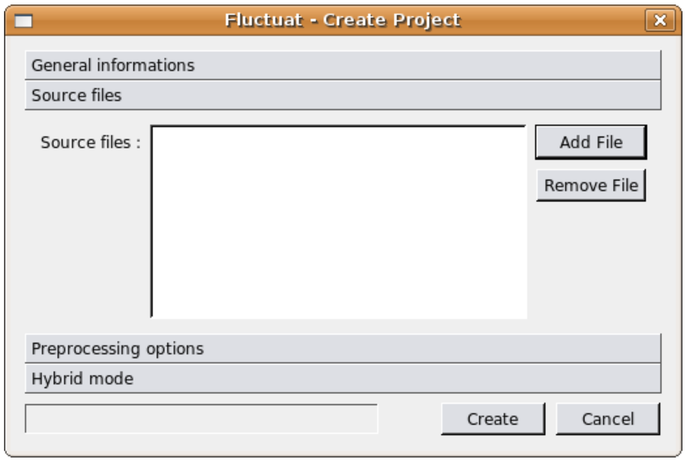


FIGURE 2 – Project creation window.

script_general_new.sh (using button **choose** at the right of the **preprocessing script** box),
- and finally, select the **Create** button, to create the project.

The source files can include almost any feature of C programs. The input/output functions, such as **printf** or **scanf**, which can not be interpreted by the analyser, are not accepted. When the project creation does encounter a problem (the source code is incorrect), the compilation errors can be seen by selecting **View Error Log** in the **Analysis** menu. Features accepted by the C language but not by the analyser, such as printf, are accepted in the creation phase, an error occurs only in the analysis phase.

A number of assertions can be added to the analysed C source code, to help the analyser through interactions with users, to give some hypotheses on the programs, or to print information on evolution of variables in the program, and not only at its end.

2.3 Assertions in the source code

We first detail the assertions that allow to give information on variables, then those used to print information on evolution of variables in the program. These assertions can be simply inserted by hand in the source code. All assertions have names that specify the type of the variable considered (names including, depending on the type of assertions, D for double precision, F for floating-point precision and I for integers, or the complete keywords DOUBLE, FLOAT, and INT).

2.3.1 Specifying ranges on values of variables

Wen we specify a variable only by a range of value, we consider there is no representation error, the given range is thus the range of both floating-point and real value of the variable.

We can first declare that a variable x has a bottom (empty) value. It can be used if the user knows or wants the analyser to know for sure that in some branch of execution, x is not assigned. It can also force the analyser (if this is done for all variables known at this location of the program) to consider the sequel of the program as dead code. Note the difference in the assertions used, depending on the type (double, floating-point or int) of the variable set.

```
double x;
float y;
int i;
```

```
.....
x = __BUILTIN_DAED_DBOTTOM;
y = __BUILTIN_DAED_FBOTTOM;
i = __BUILTIN_DAED_IBOTTOM;
.....
```

On the other side of the spectrum, one can declare :

```
double x;
float y;
int i;
.....
x = __BUILTIN_DAED_DTOP;
y = __BUILTIN_DAED_FTOP;
i = __BUILTIN_DAED_ITOP;
.....
```

This assigns abstract value "top" ($\top$) to variable x, y and i. This forces Fluctuat to analyse the behaviour of a C program for all possible values of (input) variables x, y and i. This can in particular be used as a stub for some functions. For instance,

```
double x;
scanf("%e",&x);
```

could be replaced by the following piece of code :

```
double x;
x=__BUILTIN_DAED_DTOP;
```

Now, if we want to declare a set of possible input values instead of one value for a variable, we can use :

```
float x;
.....
x=__BUILTIN_DAED_FBETWEEN(a,b);
y=__BUILTIN_DAED_DBETWEEN(c,d);
i=__BUILTIN_DAED_IBETWEEN(e,f);
.....
```

This tells the analyser that, respectively, floating-point variable x is between a and b ; double precision variable y is between c and d and finally, integer value i is within e and f. All these values have no imprecision error. It can also be used to specify that a constant has no error :

```
float x;
.....
x=__BUILTIN_DAED_FBETWEEN(0.333333,0.333333);
.....
```

This forces the analyser to consider that the floating-point representation of .333333 (slightly different from decimal number .333333) is the intended real number.

2.3.2 Specifying ranges on values and errors of variables

Now we want to specify also a representation error (between real and floating-point value) to a variable. We thus have to decide if the value set is the floating-point or the real value.

– In the non-relational analysis mode, the analysis follows the floating-point control flow and deduces the real value from addition of the floating-point value and the error. It is thus more natural to use assertions on the floating-point value and errors.
– In the relational analysis modes, the analysis uses relations on the real value and errors, and deduces the floating-point value from the real value and error, it is thus more natural (and accurate) to use assertions on the real value and errors.

However, all assertions can still be used with both kinds of analyses.

Assertions specifying the floating-point value and error.
Because real numbers can not always be exactly represented by floating-point numbers, one may want the analyser to understand that only an approximation of a constant is known, and that the error committed by this approximation is bounded by the ulp of this constant :

```
double x;
float y;
.....
x = __BUILTIN_DAED_DBETWEEN_WITH_ULP(a,b);
y = __BUILTIN_DAED_FBETWEEN_WITH_ULP(a,b);
.....
```

Variable x has a floating-point value in $[a, b]$, and it may have an initial rounding error bounded by $[-ulp(c), ulp(c)]/2$, where $c = \max(|a|, |b|)$.

One can also specify an initial error together with a range of values for each variable :

```
double x;
float y;
int i;
.....
x = __BUILTIN_DAED_DOUBLE_WITH_ERROR(a,b,c,d);
y = __BUILTIN_DAED_FLOAT_WITH_ERROR(a,b,c,d);
i = __BUILTIN_DAED_INT_WITH_ERROR(a,b,c,d);
.....
```

The analyser now knows that variable x has floating-point values between constants a and b and error (localised at the line at which the assertion has been written) between constants c and d. The real value of x has its value between $a + c$ and $b + d$. For variable y, this is the same, except y is in double precision. The same can be specified for an integer i, with integer bounds a, b, c and d.

Instead of the absolute error, a maximum relative error (in absolute value) can also be specified together with the range of the floating-point value :

```
double x;
float y;
.....
x = __BUILTIN_DAED_DOUBLE_WITH_RELERROR(a,b,c);
y = __BUILTIN_DAED_FLOAT_WITH_RELERROR(a,b,c);
.....
```

In the code above, **x** is given a floating-point value within the interval **[a,b]**, with a potential error with is between **-c** and **c** times the maximum absolute magnitude of numbers in this interval of floating-point numbers. A more detailed exampled can be found in **EXAMPLES/testeps10**.

There are two main practical uses of these keywords. First, it is useful in order to specify environments in which to analyse some programs : typically external inputs with some known imprecision errors (see examples **stability1** and **stability2** for instance, Section 3.2.1). This can also be used for simulating by hand a modular analysis. Secondly, it can be used for specifying the imprecision at which some constants have been written in the program.

Assertions specifying the real value and error.

```
double x;
float y;
int i;
.....
x = __BUILTIN_DAED_DREAL_WITH_ERROR(a,b,c,d);
y = __BUILTIN_DAED_FREAL_WITH_ERROR(a,b,c,d);
i = __BUILTIN_DAED_IREAL_WITH_ERROR(a,b,c,d);
.....
```

We can now specify to the analyser that a variable x takes its real value between constants a and b, and an error (localised at the line at which the assertion has been written) in $[-d,-c]$, so that the floating-point value of x takes its value between $a+c$ and $b+d$.

Example.

```
double x, y, z;
  x = __BUILTIN_DAED_DBETWEEN(-400,100); /* real or float value (no error) */
  y = __BUILTIN_DAED_DOUBLE_WITH_ERROR(-400,100,-10,0.1);
  z = __BUILTIN_DAED_DREAL_WITH_ERROR(-400,100,-10,0.1);
```

After this sequence of assertions, the analyser prints via the graphic interface the following results for x, y and z :

variable	real value	floating-point value	error
x	[-400,100]	[-400,100]	[0,0]
y	[-410,100.100001]	[-400,100]	[-10,0.1]
z	[-400,100]	[-410,100.1]	[-0.1,10]

2.3.3 Limiting ranges of values and errors of variables

One may also want to no longer specify ranges of values and errors of inputs, but limit the ranges of values and errors of existing variables. In the same way as above, there are assertions allowing to limit the ranges of floating-point or real values, and it is more natural and accurate to limit the range of the real value in the case of relational analyses.

The following code specifies to the analyser the fact that the floating-point value range of variable $x2$ is obtained by intersecting the range of floating-point value of variable $x1$ with the interval $[a,b]$, and its error is the error of variable $x1$. The value range of $x2$ may thus not take the full range $[a,b]$. This is done typically for helping the analyser when it is not precise enough.

```
double x1, x2;
```

```
float y1, y2;
int i1, i2;
.....
x2 = __BUILTIN_DAED_LIMIT_DOUBLE(x1,a,b);
y2 = __BUILTIN_DAED_LIMIT_FLOAT(y1,a,b);
i2 = __BUILTIN_DAED_LIMIT_INT(i1,a,b);
.....
```

The same assertions, but for limiting the ranges of the real value of variables write

```
double x1, x2;
float y1, y2;
int i1, i2;
.....
x2 = __BUILTIN_DAED_LIMIT_DREAL(x1,a,b);
y2 = __BUILTIN_DAED_LIMIT_FREAL(y1,a,b);
i2 = __BUILTIN_DAED_LIMIT_IREAL(i1,a,b);
.....
```

To limit the ranges of the error on a variable, we can write

```
double x1, x2;
float y1, y2;
int i1, i2;
.....
x2 = __BUILTIN_DAED_LIMIT_DERROR(x1,a,b);
y2 = __BUILTIN_DAED_LIMIT_FERROR(y1,a,b);
i2 = __BUILTIN_DAED_LIMIT_IERROR(i1,a,b);
.....
```

Used in the non-relational mode, $x2$ takes the same floating-point values as $x1$, and its error is limited by the range $[a, b]$. The real values are in $[\min(x1) + a, \max(x1) + b]$.
Used in a relational mode, $x2$ takes the same real values as $x1$, and its error is limited by the range $[-b, -a]$, so that the floating-point values are in $[\min(x1) + a, \max(x1) + b]$.

In the case when the bounds are actually used to limit the existing error, the drawback of the use of this assertion may be that the provenance (in terms of lines in the C source code) of the errors is lost, the range of error is associated to the line of the assertion.

Assertions also exist, that limit both the ranges of floating-point values and errors :

```
double x1, x2;
float y1, y2;
int i1, i2;
.....
x2 = __BUILTIN_DAED_LIMIT_DFLOAT_AND_ERROR(x1,a,b,c,d);
y2 = __BUILTIN_DAED_LIMIT_FFLOAT_AND_ERROR(y1,a,b,c,d);
i2 = __BUILTIN_DAED_LIMIT_IFLOAT_AND_ERROR(i1,a,b,c,d);
.....
```

For example, variable $x2$ is assigned to a floating-point value equal to that of $x1$ intersected with $[a, b]$, and its error is assigned to that of $x1$ intersected with $[c, d]$, so that the real value of $x2$ lies in a sub-interval of $[a + c, b + d]$.

Finally, assertions that limit the ranges of real values and errors are

```
double x1, x2;
float y1, y2;
int i1, i2;
.....
x2 = __BUILTIN_DAED_LIMIT_DREAL_AND_ERROR(x1,a,b,c,d);
y2 = __BUILTIN_DAED_LIMIT_FREAL_AND_ERROR(y1,a,b,c,d);
i2 = __BUILTIN_DAED_LIMIT_IREAL_AND_ERROR(i1,a,b,c,d);
.....
```

For example, variable $x2$ is assigned to a real value equal to that of $x1$ intersected with $[a, b]$, and its error is assigned to that of $x1$ intersected with $[-d, -c]$, so that the floating-point value of $x2$ lies in a sub-interval of $[a + c, b + d]$.

Example.

```
 double y1,y2,y3,y4,y5,y6;

 y1 = __BUILTIN_DAED_DOUBLE_WITH_ERROR(-400,100,-10,0.1);

 y2 = __BUILTIN_DAED_LIMIT_DREAL(y1,0,50);
 y3 = __BUILTIN_DAED_LIMIT_DREAL_AND_ERROR(y1,0,50,-100.0,0.0);
 y4 = __BUILTIN_DAED_LIMIT_DOUBLE(y1,0,50);
 y5 = __BUILTIN_DAED_LIMIT_DOUBLE_AND_ERROR(y1,0,50,-100.0,0.0);
 y6 = __BUILTIN_DAED_LIMIT_DERROR(y1,-100.0,0.0);
```

After this sequence of assertions, the analyser prints via the graphic interface the results summed-up in the following table (in which, when the results are the same for relational and non-relational analyses, the type of analysis is not specified) :

variable	real value	floating-point value	error
y1	[-410,100.100001]	[-400,100]	[-10,0.1]
y2 (non-relational)	[-10.1000001,60.1000001]	[-0.1,60]	[-10,0.1]
y2 (fully relational)	[0,50]	[-0.1,60]	[-10,0.1]
y3 (non-relational)	[-0.100000001,50.1000001]	[-0.1,50]	[0,0.1]
y3 (fully relational)	[0,50]	[-0.1,50]	[0,0.1]
y4	[-10,50.1]	[0,50]	[-10,0.1]
y5	[-10,50]	[0,50]	[-10,0]
y6 (non-relational)	[-410,100]	[-400,100]	[-10,0]
y6 (fully relational)	[-410,100.100001]	[-400,100]	[0,0.1]

Indeed, y2 keeps the same error as y1 in [-10,0.1], and its real value is limited in $[0, 50]$. The floating-point value is then obtained by $([0, 50] - [-10, 0.1]) \cap [-400, 100] = [-0.1, 60]$. However, it is the logics of the non-relational analysis that the real value printed is the sum of the range of the floating-point value and the error, thus finally giving as real range $[-0.1, 60] + [-10, 0.1] = [-10.1, 60.1]$. Whereas in the fully-relational mode, the real value printed is the actual one. This shows why the assertions limiting the real values of variables should be avoided in the non relational mode.
In the same way, the real range of $y3$ is first truncated to $[0, 50]$, the error is limited by $[0, 100]$, which gives $[0, 0.1]$, and the floating-point value in non relational mode is got by $[0, 50] - [0, 0.1] = [-0.1, 50]$. Same as before, we then print not the actual real range, but that obtained by addition of floating-point range and error, which is $[-0.1, 50.1]$.
When the assertion is set on the floating-point value, the result of the assertion is the same in relational or non-relational mode (see $y4$ or $y5$). In the assertion limiting the floating-point value and error of $y5$, note

that the error is here limited by $[-100, 0]$.
When we limit only the error, in the non-relational mode, we keep for the floating-point value of $y6$ that of $y1$, $[-400, 100]$, and limit the error by $[-10, 0.1] \cup [-100, 0] = [-10, 0]$. This gives a real value in [-410,100]. In the relational mode, we keep the real value of $y1$, $[-410, 100.1]$, and limit the error by $[-10, 0.1] \cup [0, 100] = [0, 0.1]$. The result if of poorer quality than the non-relational one, because the values and errors were set on $y1$ with an assertion on floating-point value and error, which is better suited to the non-relational mode.

2.3.4 Assertions on the dynamics of the inputs of a system

In some cases, bounds on the values are not sufficient to describe accurately the behaviour of a system : we thus propose an assertion that allows to bound, in a loop indexed by an integer variable **i**, the variation between two successive values of an input variable **x** :

```
double x;
.....
  for (i=i0 ; i<N ; i++) {
    .....
    x = __BUILTIN_DAED_FGRADIENT(x0min,x0max,gmin(i),gmax(i),xmin,xmax,i,i0);
    .....
  }
.....
```

In this assertion, $i0$ is the value of variable **i** at first iteration. The value of **x** at first iteration is in interval $[x0min, x0max]$, the difference between two successive iterates is in the interval $[gmin(i), gmax(i)]$, which bounds may depend on the iterate, and the value of **x** is always bounded by $[xmin, xmax]$. Thus $x(i0) = [x0min, x0max]$, and for all $i \in \{i0, \dots, N\}$, we have

$$x(i) = (x(i-1) + [gmin(i), gmax(i)]) \bigcap [xmin, xmax].$$

Our relational domain (section 4) is specially well adapted to dealing with these relations between iterates in a loop.

2.3.5 Join and meet assertions

These two keywords allow for union or intersection of values and errors :

```
double x,y,z,t;
.....
x=__BUILTIN_DAED_DJOIN(z,t);
y=__BUILTIN_DAED_DMEET(z,t);
.....
```

These take respectively the union of values and errors of z and t and put it in x, and the intersection of values and errors of z and t and put it in y (in the example here, these assertions are defined for double variables, but there exist equivalents for float or integer variables.

As a practical use, the user can help the analyser (using the intersection operator) or write stubs for some functions, for which we do not have the code or that we do not want to analyse (see for instance example "E3", Section 3.5.2).

Also, the JOIN assertion can be used for improving the precision of analysis by a local and possibly irregular subdivision, coded in the program to analyse, see section 4.7. In some cases when the result of interest is the variable that is computed using this JOIN assertion, and relations between this variable and inputs are not needed for further computations, the assertion __BUILTIN_DAED_DCOLLECT can be used instead of __BUILTIN_DAED_DJOIN. With DCOLLECT, non relational union are made between the different abstract values that are joined, which may be much more efficient in time.

2.3.6 Assertions to print information during the analysis

All information displayed by default by the interface, is about the state of variables at the end of the program. If the user wants to visualise information during the analysis, he must specify which variable he wants to see, and at which place in the program.

This is done for a floating-point variable **S** for instance, by inserting an assertion

__BUILTIN_DAED_FPRINT(S)

at some places in the program. Of course, the corresponding assertions **__BUILTIN_DAED_DPRINT(S)** and **__BUILTIN_DAED_IPRINT(S)** exist for double precision floating-point and integer variables. Then, the analyser will print in a file, the interval value and global error for this variable, each time it meets the assertion. For example, if it is in a loop, it will print the value at each fixpoint iteration. Some buttons in the user interface then allow to draw figures representing this evolution (see section 2.5).

Also, when the weakly-relational mode of the analyser (see sections 2.4 and 4) is selected, a more complete representation of the values of variables can be printed using assertion **__BUILTIN_DAED_FAFFPRINT(S)**. Note that the use of these assertions has changed, and that the older one (for example **S** = **__BUILTIN_DAED_DPRINT(S,S)**) is no longer supported.

2.3.7 Assertions to retrieve input-output relations

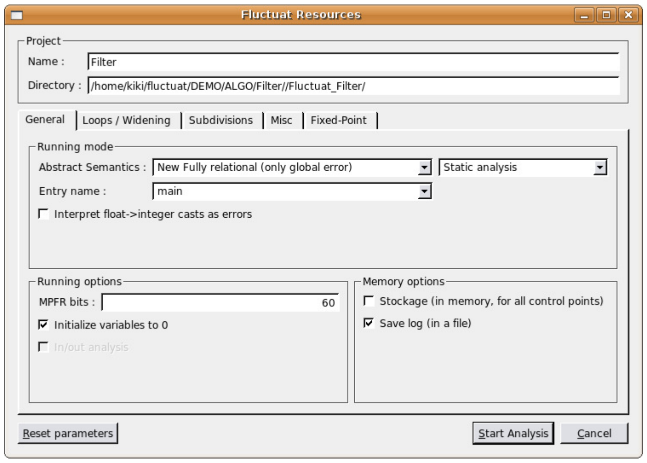
2.3.8 Simplified aliases for the assertions

We are aware that the length of current names of the assertions is quite tedious. That is why we defined aliases to all previous assertions that allow to omit the **__BUILTIN_DAED_** part. For example, we can also set the range of a double precision variable x in $[0, 100]$ by

```
x = DBETWEEN(0,100);
```

2.4 Static analysis : parameters

When we select the **Start Analysis** button, the window shown in figure 3 appears. The parameters which tune the precision of the analysis, can then be edited (default values are given for each parameter). There



Project
Name : Filter
Directory : /home/kiki/fluctuat/DEMO/ALGO/Filter//Fluctuat_Filter/

Running mode
Abstract Semantics : New Fully relational (only global error) | Static analysis
Entry name : main
☐ Interpret float->integer casts as errors

Running options
MPFR bits : 60
☑ Initialize variables to 0
☐ In/out analysis

Memory options
☐ Stockage (in memory, for all control points)
☑ Save log (in a file)

FIGURE 3 – Parameters of the fluctuat analyser.

are several sub-menus which we describe here (see appendix for use in batch mode). The boxes **Project Name** and **Project Directory** are accessible from each sub-menu, and their content can be changed either by hand, or using the **Choose Project** button (also see section 2.1). The **Reset** button allows to set all parameters (not just the ones defined in the current sub-menu) back to their default values.

We now describe one by one the parameters that can be modified in each tab of the parameters window :

2.4.1 General

- **Abstract semantics** : one can choose between three semantics for the analysis. The default one, **non-relational abstract domains**, derives naturally from the abstraction with intervals, of the concrete semantics described in the introduction. But it suffers from the drawbacks of interval arithmetic, that is the wrapping effect due to the lack of relations between variables in the lattice. Thus, a mode **weakly-relational abstract domains** is also proposed, but not fully implemented yet. Finally, choosing **fixed-point semantics** selects the mode for the analysis of fixed-point implementations of floating-point programs, with the same general principle of computing the value and a sum of errors.
- **Static Analysis / Symbolic Execution** : instead of computing fixpoint by unions over iterations, the analyzer unrolls completely the loops. Input variables must have as value an interval with zero width, but can be perturbed by interval errors. If input variables have their value in an interval with non zero width, the center of the interval is taken as value, and a warning message is issued (that can be seen at the log console).
  When this option is chosen, some parameters in the **Loops** and **Widening** sub-menus can not be set. Neither can the **Automatic Parameters** option.
- **Entry Name** : the name of the function to analyse in the code (by default, **main**) - of course, all functions called from it are analysed.
- **MPFR Bits** : integer parameter, that defines the number of bits used for MPFR (internal representation of real numbers used by the analyser). This number should in most cases be greater than the number of bits used for the representation of double precision numbers, that is 53 bits (or 25 bits if only simple precision floating-point variables are used in the C code to analyse).
- **Initialise variables to 0** : enables to specify if the variables which are non-initialised have value **top** or **0**.
- **In/out analysis** : for the time being, this option is possible only with the non relational domain (this latter is automatically selected when the in/out analysis is asked). The idea is that for large programs, the user may want to make a first fast analysis to determine the in and out status of variables, for example to check that all variables are initialised. And then he can launch a more accurate analysis.
- **Stockage** When the box is not ticked, the storage is minimum, the states of variables are kept only at control points where they are necessary to the analysis. Otherwise, they are kept at any point of the program (needs more memory space).
- **Save log** Keep activated if you want to keep all the informations about the analysis. Be carefull, the log file can be huge in some cases (with subdivisions as example).

2.4.2 Loops / Widening

The widening operators allow to extrapolate the least fixed point computation after some number of iterations. They are for the time being used on all variables at the same time. We propose two types of widening operators :

- the classical widening operator, that extrapolates towards infinity if the sets of values are still growing after some iterations, and is mandatory, as it ensures that the analysis terminates in finite time.
- the widening we call **progressive widening**, which if selected, is called before the classical widening operator, and that tries to approximate as closely as possible the least fixed point by successive approximations, using a progressive reduction of the precision of the numbers used for the analysis. For the time being, this widening is used only for the computation of the errors (not for the values).

The parameters that can be set by the user are thus the following :

- **Widening Threshold** : is an integer that indicates after how many iterations the analyser will begin to use a widening operator (either the progressive one if the option is selected, or otherwise the classical one). The bigger this parameter is, the more precise (and time consuming) the analysis gets. When a loop is unrolled, the **Widening Threshold** parameter indicates how many times the new functional is executed before using widening operators. It is thus equivalent to using widening operators after **Widening Threshold * Unfold Loop Number** iterations of the real loop.

– **Narrowing Threshold** : is an integer which enables the analyser to decide after how many iterations it will extrapolate the least fixed point computation when it gets to a post fixpoint, by using a narrowing operator. There again, the bigger it is, the longer the analysis takes. Due to the narrowing operator used for the time being, this parameter is set to the default value of 2. If the user really wants to change this value (but it should not change much the result), this can be done in the batch mode. The following example explains the least fixpoint computation in the presence of classical widening/narrowing operators :

```
(1): int i=1;
(2): while (i<100)
       (3): i++; (4):
(5):
```

The abstract semantic equation for the invariant at control point (3) (beginning of the body of the loop) is :

$$I_3 = ([1,1] \cup (I_3 + [1,1])) \cap]-oo, 99]$$

Without any widening, we get the following fixpoint iteration sequence : $I_3^0 = \perp$, $I_3^1 = [1,1]$, $I_3^2 = [1,2]$,$\cdots$, $I_3^j = [1,j]$, $\cdots$, $I_3^{100} = I_3^{99} = [1,99]$. With a very classical widening operator, after $j << 100$ steps, we get $I_3^{j+1} = [1,+oo[$. Then, $I_3^{j+2} = [1,99]$ (no need for narrowing here !).

Then, if the progressive widening is selected by ticking the corresponding box, three more parameters are taken into account :

– **Loss of precision :** sets how many bits of precision must be lost at each iteration
– **Minimal precision :** sets the minimum number of bits of precision we want for representing errors (when it is reached, we iterate the fixed-point computation with same precision until convergence or reaching the standard widening threshold).
– **Threshold before standard widening :** it may be that this progressive widening also converges too slowly, so a bound is given for the number of iterations using the progressive widening, after which the standard widening will be used.

The first two parameters are common to all types of analysis (relational or not, floating or fixed-point semantics), but are accessible only if the **Symbolic Execution** mode is not selected in the (**General** sub-menu). Also, when the **Automatic Parameters** mode is selected in this same sub-menu, the **Initial Unfolding** parameter is not accessible, and the unfolding and widening thresholds are set only for the computation of values (and not errors).

– **Initial Unfolding** : positive or zero integer, that defines the number of time the loop is executed before starting unions over iterations (with or without unfolding). This initial execution often improves the accuracy of the result by allowing to pass over the often different behaviour of the first iterations of the loop.
– **Depth of inter-loop relational history** : is an integer that parameterises the depth of relations wanted between loop iterations. Keeping relations between values at all iterations of loops would be too costly in memory and computing time, but on the contrary keeping none leads to imprecise results in the case of iterative schemes of order greater than one. When this parameter is set to -1 (which is the default value), all relations are kept in the loop. This parameter is completely independent of the unfolding of loops.
– **Cyclic unfolding** is an integer, which enables to get more precision on loops by virtually unrolling them. Most of the time, using the initial unfolding is sufficient for getting accurate results, but in some cases unfolding can be complementary.
Consider the following program :

```
int i;
float t=1;
```

```
for (i=0; i<20; i++)
  t=t*.618
```

With **Unfold Loop Number** equal to 1, it is equivalent to the same program. When it is equal to 2 it is equivalent to

```
int i;
float t=1;
for (i=0; i<10; i++)
  { t=t*.618;
    t=t*.618;  }
```

See Section 5.3 and examples for more details.

- **Keep relations at the end of loops** : when the box is ticked, all relations computed in the loops are kept, otherwise they are agglomerated at the exit of the loop (the idea being that some relations needed in the loop for good results are no longer necessary after the loop). In the case when these relations are not kept, the computation will generally be less costly but less precise.
- **Unfold non-relational errors** : when the box is ticked, more information is computed about the iteration in loops where the errors occur - more costly with same precision, but more information may be displayed at the user interface (in the future ;-)
- **Tests interpretation on float only (daedres_floatflow)** : used only with the latest relational mode only. If value is 0 (default value), tests are interpreted on floating-point and real values, and the result is the intersection of the two (relying on the idea that the floating-point and real value control flow are the same, otherwise a warning for an unstable test is issued). If value is 1, tests are interpreted only on floating-point values. With value 0, in the general case more accurate results are obtained than when ticking the box (value 1) in order to interpret tests (and narrow values) only on floating-point numbers. But in some cases this may not give the expected results. See section 4.6 for examples of the use of these two modes.
- **Automatic Parameters** : in this mode, the analyser tries to choose "optimal" values for the number of executions before unions and number of unfolding of the loops, for each loop computation, that should allow to get tight ranges for the errors. When this parameter is set, the user can choose the **Maximum Iterations** parameter. Also some the parameters in the **Loops** sub-menu (except for **initial unfolding**) can still be set, they are those used for the computation of the values of variables.
- **Maximum Iterations** : integer parameter, setting the maximum number of iterations before deciding a choice of unfolding in the automatic mode. This option is available only when the **Automatic Parameters** mode is set.

These two options may allow a better convergence of the fixed-point computation in some cases, but as it is not always the case, the choice is left to the user :

- **Exact error calculation for constants** : when the box is ticked, the rounding errors are always computed as accurately as possible : when the floating-point result of an operation is a floating-point number (and not an interval), the error can thus be bounded taking as reference the same computation with higher precision floating-point numbers (MPFR library). Otherwise (if the box is not ticked), this is still the case outside loops, but for the computation or propagation of errors in loops, we use the **ulp** error, which is less accurate but often allows a better convergence of fixpoint algorithms.
- **Convergence of higher order errors** : when the box is ticked, the convergence in fixpoint computations is decided on each first-order error and the higher order error. But it has been noted that higher order errors often seriously delay the convergence of fixpoint computation, while being completely negligible. For this reason, we propose a convergence criterion based on each first-order error and the global error. The convergence is better, but higher order errors are over-approximated. The semantics of operations would have to be modified to suppress this inconvenience.

2.4.3 Subdivisions

The options proposed here for the time being are used to improve the accuracy of results, by subdividing intervals to reduce the problem of lack of correlations in the analysis. These subdivisions may be useful with both the relational and the non-relational analyses. Two different and independent ways of using subdivisions are proposed :

- **Use subdivisions for the whole program** : using large intervals as inputs can lead to large overapproximations of the results, even with the relational abstract domains (particularly when non affine operations are used, divisions, high degree polynomials, etc). Thus, if the user still wants to analyse a program for a large set of inputs, we give the possibility to use the interface to subdivide the input interval, execute as many partial analyses as the number of subdivisions indicates, and present as final result the union of all these analyses. When this option is selected, we give the user the possibility to divide at most two intervals. Indeed, the number of analyses is then the number of subdivision of first interval multiplied by the number of subdivisions of the second interval, in order to take all possibilities into account.
  The intervals to be subdivided can be selected either by giving the number of the line in the source code (beware, the first line of the source code is noted as line number 2). Or, more simply by clicking on the **Choose line** button, and selecting the line in the program by clicking.
  For the future, we think of trying to detect automatically which interval most needs subdivision, and subdivide this one. Also, intervals are for now regularly subdivided, which is not optimal, and will be improved in the future.
- **Refinement of bitwise operations** : we have noted that the propagation of existing errors through bitwise operations could lead to huge overapproximations when the interval values for the inputs have an important width. We thus give the possibility to subdivide these values for the computation of errors (by an integer number of subdivisions), the results are more accurate, but the cost in memory and computation time can be heavy. However, the subdivision is only local to bitwise operations, and only one analysis is performed.

2.4.4 Misc

CrossPlatform The choice of the number of bits used for the representation of each integer type (signed or unsigned **char**, **short** or **int**), is left to the user, except for **long int**, which are necessarily represented on 32 bits. Also, the **int** types must be represented on at least as many bits as **short** types, which must themselves be represented on at least as many bits as **char** types. This property is managed by the user interface.

Levels of abstraction One of the three following levels of abstraction for aliases must be chosen :

- **No pointers** : aliases (arrays, pointers) are not interpreted. This option must be preferred to the others when the programs being analysed contains no alias, because the analysis is quicker and uses less memory. One must be careful : if ticking this box when a programs does contain aliases, the analysis may produce no error, while giving false results.
- **Abstract arrays as one location** : aliases are interpreted, but only one value agglomerates all coefficients of an array. However, the fields of structures are not agglomerated. The analysis may be unprecise in presence of arrays.
- **Unravel arrays** : aliases are interpreted, and one value is associated to each coefficient of an array. The analysis may be costly.

Interprocedural

- **Do not limit call stack** : when the box is not ticked, the size of the call stack is limited by the integer **Call Stack Depth**. Otherwise, it is not limited, but only the last call is kept in memory (indeed, the others are not needed if we are interested mainly by the result at the end of the program). The results are most of the time more accurate and got faster, but limiting the call stack is still useful in some cases, for analysing recursive functions for example.

– **Call Stack Depth** : An integer which should not be too big. It describes the size of the (simple) abstraction of the call stack. For instance, in the following code :

```
float f(float x)
{  ...
   (1): y=g(x);
   ...
   (2): z=g(2*x);
   ...  }

float g(float z)
{  ...
   (3): t=h(z);
   ...
   (4): u=h(z/2);
   ...  }
```

– there are 4 instances of **h** : (1,3), (1,4), (2,3) and (2,4),
– (1,3) and (2,3) are “unioned” (identified) if the call stack depth is 1,
– the 4 instances are analysed separately if the call stack depth is greater or equal to 2.

As a consequence, the bigger the call stack depth, the more precise the analysis gets, since functions are then analysed separately for different contexts. The analysis gets also more costly in terms of time and memory consumption. Of course, this parameter can be set only if the box **Do not limit call stack** is not ticked.

2.4.5 Fixed-Point

A new mode for analysing the errors that would be committed with a C program, but using fixed-point numbers is proposed. Variables of the analysed program are still defined as double or float variables, but treated with a fixed-point behaviour, using assertions specifying the format of each variable, with notations much like those of System C. The parameters and assertions are not detailed yet.

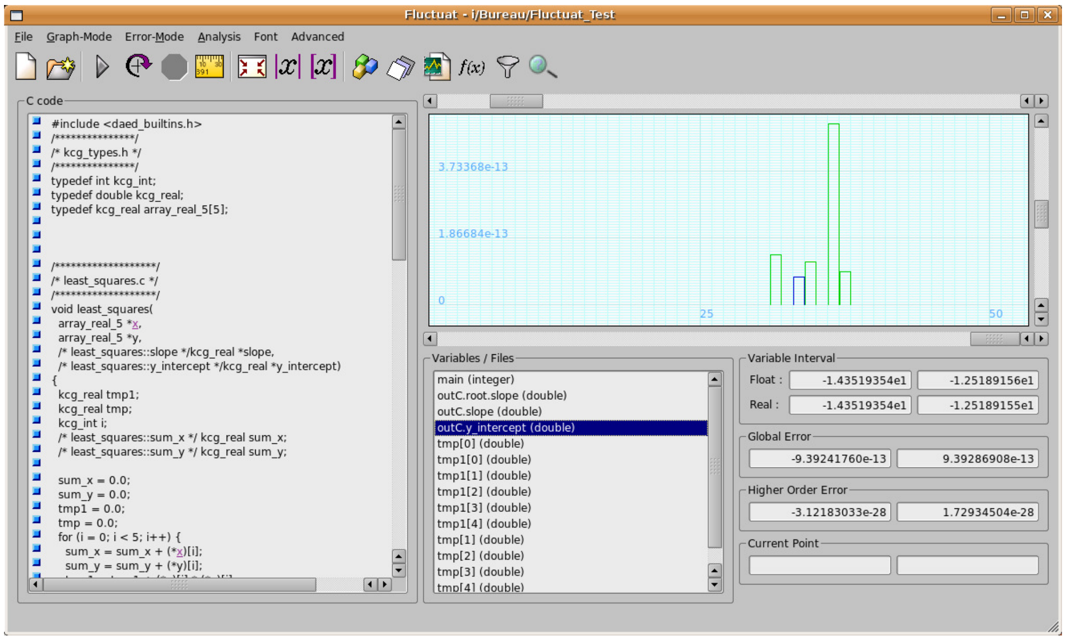
Also, a program called **autofix**, relying on the analyser, was designed to determine, starting from a floating-point program, the smallest fixed-point format of all variables that would be needed to get the same order of magnitude of errors at the end of the program as with floating-point program. We get indeed the smallest format when there are no cyclic dependencies between variables, and otherwise a format that satisfies the constraint on the errors but may not be optimal. This program is not available through the user interface for the time being.

2.5 Visualisation of the results

As shown in Figure 4, the main window of the analyser displays the code of the program being analysed (left hand side), the list of identifiers (bottom, right hand side of the main window) occurring in the abstract environment at the end of the analysis and a graph representation (right hand side of the main window) of the abstract value related to the selected identifier in the list. The list of identifiers is automaticly sorted in alphabetic order. The graph represents the error series of a variable *id* and thus shows the contribution of the operations to the global error on *id*. The operations are identified to their program point which is displayed on the X-axis. Scrollbars on the sides of the graph window are used to do various kinds of zooms. A **Reset Graph** button allows to come back to the default places of the scrollbars.

Reset graph button

In Figure 4, the bars indicate the maximum of the absolute values of the interval bounds. This enables to assert the correctness of the code, when the errors have a small magnitude. The graph and program code



```
#include <daed_builtins.h>
/****************/
/* kcg_types.h */
/****************/
typedef int kcg_int;
typedef double kcg_real;
typedef kcg_real array_real_5[5];



/*******************/
/* least_squares.c */
/*******************/
void least_squares(
  array_real_5 *x,
  array_real_5 *y,
  /* least_squares::slope */kcg_real *slope,
  /* least_squares::y_intercept */kcg_real *y_intercept)
{
  kcg_real tmp1;
  kcg_real tmp;
  kcg_int i;
  /* least_squares::sum_x */ kcg_real sum_x;
  /* least_squares::sum_y */ kcg_real sum_y;

  sum_x = 0.0;
  sum_y = 0.0;
  tmp1 = 0.0;
  tmp = 0.0;
  for (i = 0; i < 5; i++) {
    sum_x = sum_x + (*x)[i];
    sum_y = sum_y + (*y)[i];
```

```
main (integer)
outC.root.slope (double)
outC.slope (double)
outC.y_intercept (double)
tmp[0] (double)
tmp1[0] (double)
tmp1[1] (double)
tmp1[2] (double)
tmp1[3] (double)
tmp1[4] (double)
tmp[1] (double)
tmp[2] (double)
tmp[3] (double)
tmp[4] (double)
```

Float :	-1.43519354e1	-1.25189156e1
Real :	-1.43519354e1	-1.25189155e1
Global Error	-9.39241760e-13	9.39286908e-13
Higher Order Error	-3.12183033e-28	1.72934504e-28
Current Point		

FIGURE 4 – Main window of the fluctuat analyser.

frames are connected in the graphical user interface in such a way that clicking on a column of the graph makes the code frame emphasises the related program line. This enables the user to easily identify which piece of code mainly contributes to the error on the selected variable. Moreover, the source file is displayed in html, which allows to select the error graph of a variable by clicking on the variable of interest in the program frame.
Also, when a given variable is selected, the user can know which lines mainly contribute to its error, using the **Biggest error** and **Next error** buttons on the top of the main window. When choosing the biggest error, the corresponding line in the C program window is highlighted. Then, each time the user clicks the **Next error** button, the line corresponding to the next error in terms of maximum magnitude is highlighted.

Biggest error button

Next error button

In the example of Figure 4, a typical program of an instrumentation software is being analysed. It is basically an interpolation function with thresholds. One can see from the graph at the right-hand side of the code that the sources of imprecision for the final result of the main function are (variable "main" selected in the list of variables) : the floating-point approximation of the constant 2.999982 at line 20 (click on the first bar to outline the corresponding line in the C code), which is negligible, the 2nd **return** line 26 (second bar in the graph), and the 3rd **return** line 28 (third and last bar in the error graph), the last two ones being the more important. An error peak is blue when the largest bound of the error interval is negative, otherwise it is green. In fact, using **__BUILTIN_DAED_FBETWEEN** (an assertion of the analyser), we imposed that **E1** takes its value at the entry of the function between -100 and 100. So the analyser derives that the function can go through all **return** statements. But in the first and fourth **return**, the multiplication is by zero and the constant is exact in the expression returned, so that there is no imprecision. The user can deduce from this that if he wants to improve the result, he can improve the accuracy of the computation of the 2nd and 3rd **return**. One simple way is to improve the accuracy of the two subtractions in these two expressions (using **double E1** in particular), whereas the improvement of the precision of the constant 2.999982 is not the better way. Notice that the analyser also finds that the higher-order errors are always negligible.

From the interface, one can draw, for a variable selected in the variable list, the evolution over iterations of the value, the global error, or the global relative error. For this to be possible for a variable **S** for instance, the user has to use the assertion **S=__BUILTIN_DAED_FPRINT(S,S)** in the program, so that the values and errors of variable **S** are logged each time the analyser meets this assertion. The corresponding icons in the toolbar at the top of the main window, are :

floating-point value

real value

global error

relative global error

This will open a new window containing the relevant graphics, as shown in Figure 5. All these windows can be printed (in postcript in a file or on a printer), or saved as a bitmap file.

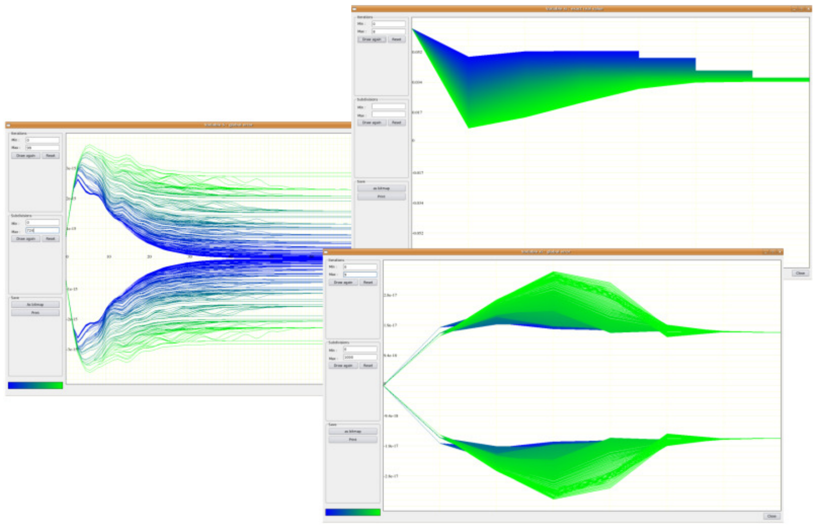
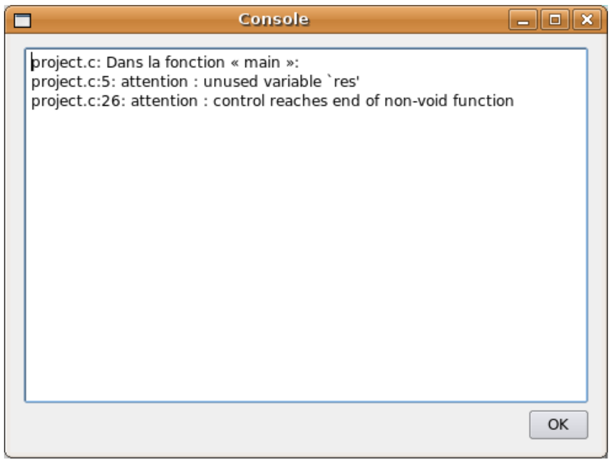


Figure 5 – Evolution of the value of a variable.

There also exists a **Font** menu : one can use small or big fonts. Big fonts come with thicker graphs on the error graph sub-window. This is convenient in particular for using the analyser during a presentation (with a video-projector).

Some information about the analysis are also available at the end, by different options in the **Analysis** menu :

– When the analysis crashes, an error message “Error during the analysis” appears. To know more precisely what happened, one can select **View Error Log** in the **Menu** menu. A more detailed error log then appears (see Figure 6).



```
project.c: Dans la fonction « main »:
project.c:5: attention : unused variable `res'
project.c:26: attention : control reaches end of non-void function
```

FIGURE 6 – Analysis error console

– During the analysis, choosing **View Monitor** opens a window giving an estimation of the progress of the analysis (see Figure 7) : the first bar gives the current proportion of instances analysed, when a

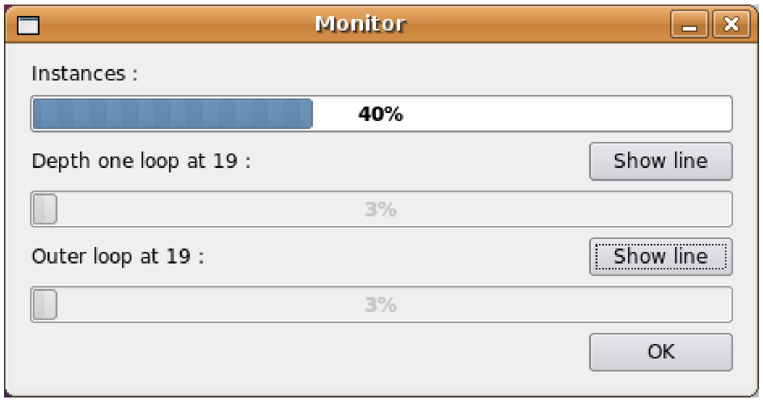
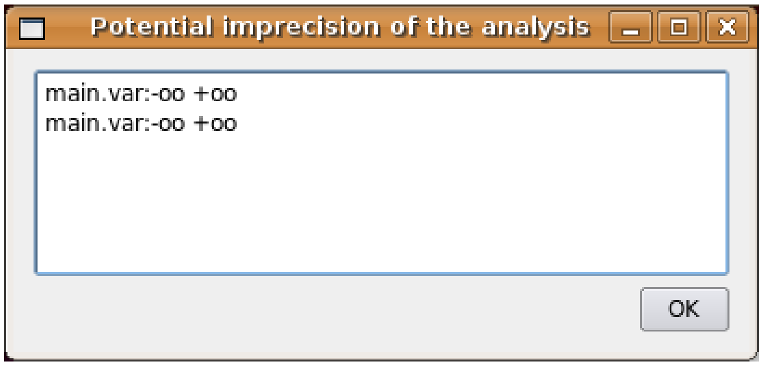


FIGURE 7 – View Monitor console

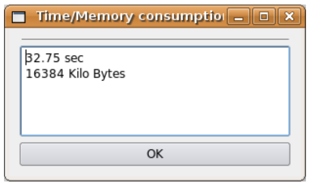
program is analysed using subdivisions of some of the input sets. The other bars indicate the current iteration of the current nest of loops, and the maximum number of iterations for each of these loops, which indicates a pessimistic bound for the remaining time needed.

– When using subdivisions, the union of results on values and errors for subdivisions already analysed can be seen via the graphic interface during the remaining analyses. Also, the first time the analyser founds a value or error on a variable possibly infinite for one subdivision, an alarm box opens (see figure 8). In this case, the user may decide the analysis will not be accurate enough. In this case, he can interrupt the analysis, by selecting **Abort Analysis** in the **Analysis** menu.
– At the end of the analysis, the time and memory needed can be seen by selecting **View Stat** (see Figure 9). Moreover, information about the analysis can be seen by selecting **View Log** (see Figure 10).
– The warnings about potential run time errors, such as division by zero or the square root of a negative number, are recalled in a separate window, by selecting **View Potential Errors** (see Figure 11 left).



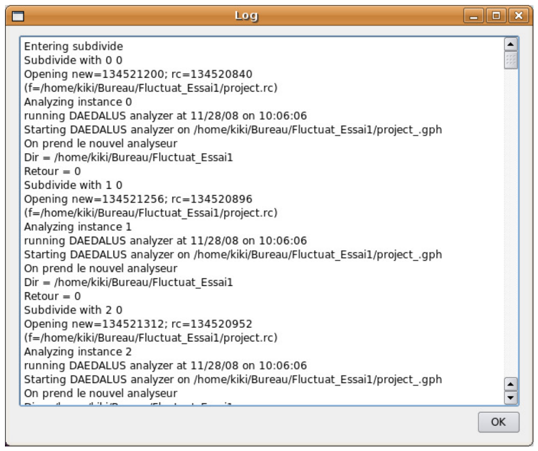
```
main.var:-oo +oo
main.var:-oo +oo
```

FIGURE 8 – Alarm window



```
32.75 sec
16384 Kilo Bytes
```

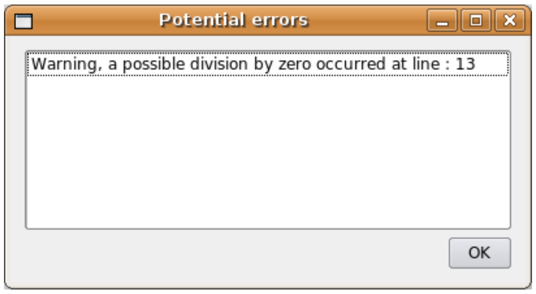
FIGURE 9 – View stat console



```
Entering subdivide
Subdivide with 0 0
Opening new=134521200; rc=134520840
(f=/home/kiki/Bureau/Fluctuat_Essai1/project.rc)
Analyzing instance 0
running DAEDALUS analyzer at 11/28/08 on 10:06:06
Starting DAEDALUS analyzer on /home/kiki/Bureau/Fluctuat_Essai1/project_.gph
On prend le nouvel analyseur
Dir = /home/kiki/Bureau/Fluctuat_Essai1
Retour = 0
Subdivide with 1 0
Opening new=134521256; rc=134520896
(f=/home/kiki/Bureau/Fluctuat_Essai1/project.rc)
Analyzing instance 1
running DAEDALUS analyzer at 11/28/08 on 10:06:06
Starting DAEDALUS analyzer on /home/kiki/Bureau/Fluctuat_Essai1/project_.gph
On prend le nouvel analyseur
Dir = /home/kiki/Bureau/Fluctuat_Essai1
Retour = 0
Subdivide with 2 0
Opening new=134521312; rc=134520952
(f=/home/kiki/Bureau/Fluctuat_Essai1/project.rc)
Analyzing instance 2
running DAEDALUS analyzer at 11/28/08 on 10:06:06
Starting DAEDALUS analyzer on /home/kiki/Bureau/Fluctuat_Essai1/project_.gph
On prend le nouvel analyseur
```

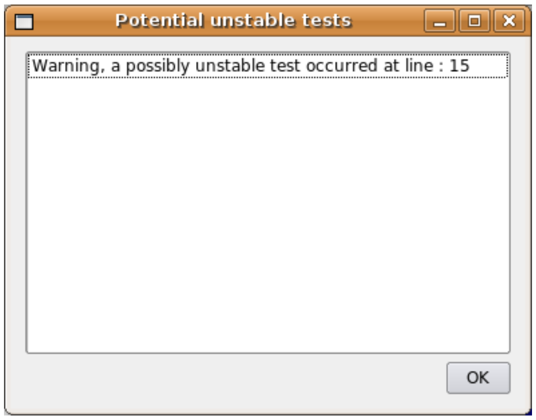
FIGURE 10 – View log console

Clicking on one error of this window selects, in the source file, the line at which this potential error occurred. The idea is not to detect run time errors, as we suppose there are none and go on with the analysis, but to point out an imprecision of the analysis which should if possible be settled, for example using assertions, in order to get more accurate results.
In the same way, the warnings about possibly unstable tests (see section 3.6) are recalled in a separate window, by selecting **View Unstable Tests** (see Figure 11 right). Clicking on one warning of this window selects, in the source file, the line at which this potentially unstable test occurred. This gives the user a summary of the assumptions under which the analysis is carried out, and allows him to check whether these unstable tests can be a problem or not.



Warning, a possible division by zero occurred at line : 13

Potential run time errors



Warning, a possibly unstable test occurred at line : 15

Unstable tests

FIGURE 11 – Warning consoles

- One can then ask which variables have possibly infinite variables (selecting **View Infinite Values**, see result in Figure 12), or possibly infinite global error (selecting **View Infinite Errors**), or possibly infinite higher order error (selecting **View Infinite Higher-Order Errors**)
- If the **In/out analysis** mode was selected when setting parameters of the analysis, one can also check the in/out status of variables. This is useful in particular to analyse parts of big programs, to check that there are no uninitialised inputs, and to spot outputs.
  When selecting **View in/out** in the **Analysis** menu, a window opens, in which all variables are printed with one of the following values :
  - **CST** (constant)
  - **initialised IN (Builtin)** (variable which first use was an initialisation using a Builtin)
  - **UNUSED** : (unused variable)
  - **non initialised but used IN** (variable used without being first initialised : a value or an assertion should be added)
  - **OUT** : any other case

  Also, variables that were used without being initialised, along with the line number of the source code at which these variables were first used, can be seen by selecting **Edit input variables** in the **File** menu.

The user can also choose, after the project creation phase, to edit the C source. In this case, he should select **Edit Source** in the **File** menu to open a new edit window, such as the one shown in Figure 13. Then the user can save, and get back to the project creation phase by clicking on the button **Done**.

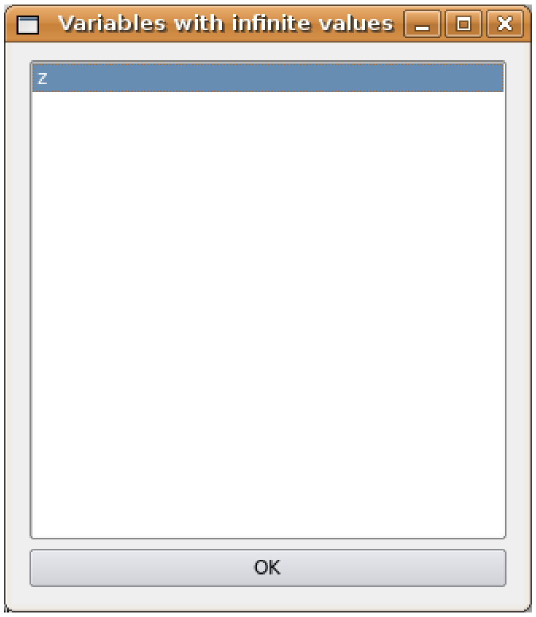


FIGURE 12 – View Infinite Values console

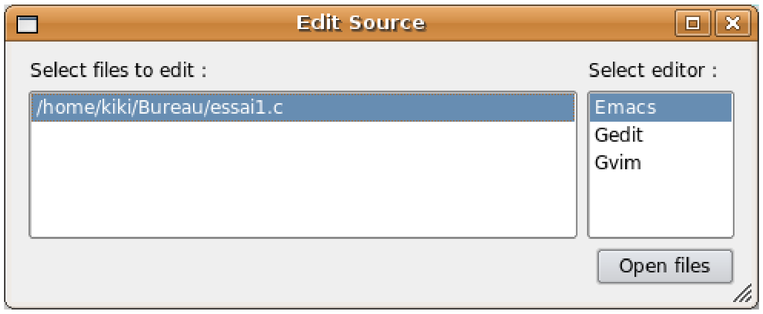


FIGURE 13 – Edit window of the fluctuat analyser.

2.6 Others

2.6.1 Edit source

2.6.2 License file

3 Non relational domain : academic examples

Academic examples are useful for two different purposes. Obviously, they help to understand how the tool works on simple examples, for which it is not too hard to guess the results. They are also of main interest to validate the tool : it is indeed difficult to validate the results provided by the analyser for large programs, since there is no way to compare them to ideal results (calculation carried out with real numbers), or to other tools. Academic examples help us check that every kind of instability is correctly detected by Fluctuat, as predicted by the theory (see for example the example computing the iterates of x^2 which only generates a catastrophic higher-order error, Section 3.2.2). The distribution includes all of these so-called academic examples.

3.1 Simple code

To begin with, just try (see EXAMPLES/testeps9/testeps9.c) :

```
#include <daed_builtins.h>

void main()
{
  double x;
  x = __BUILTIN_DAED_DOUBLE_WITH_ERROR(1,2,-0.01,0.02);
}
```

and click on x : there is a bar (with maximum value 0.02) in the corresponding histogram. When you click on it, you see the following line selected on the source part of the window :

```
  x = __BUILTIN_DAED_DOUBLE_WITH_ERROR(1,2,-0.01,0.02);
```

telling you that this error comes from the computation at this line.

Now you can also try programs like the following one (see EXAMPLES/SQRT/SQRT.c), which contains calls to external "standard libraries", such as some functions from **math.h** :

```
#include "math.h"
#include "daed_builtins.h"

int main(void)
{
  float x = 2.999982;
  float a = 0.4;
  float b = 0.8;
  float c, d, e;

  a = __BUILTIN_DAED_FBETWEEN(0,1);
  c = sqrt(a + b);
  e = c - 2*b;
  d = fabs(e);
  if (x > a + b)
     x = 0;
}
```

For the time being, the implementation of **sqrt** is very pessimistic around 0, this will be ameliorated in a further version of the tool. Notice in this example that there is an imprecision error associated with the absolute value, due to a conversion from the result (a **double**) to a **float**.

In case the analysed program does not terminate, the list of variables available at the end of the program contains only **main**, with interval value bottom ($[+\infty, -\infty]$) for value and errors. A message is also printed on the command line. See for example the program (EXAMPLES/boucle_sansfin/boucle_sansfin.c)

```
int main(void)
{
  int i;
  while (1)
    i++;
}
```

3.2 Fixpoint computations in loops

3.2.1 Stability (order 1)

Try program EXAMPLES/gold1/gold1.c with parameters,
– standard parameters first (no unfolding, **wideningthreshold** equal to 20)
– complete unfolding (**unfoldloopnumber** greater or equal to $11 = 20/2 + 1$)
In the first case, for variable **z** you see two potentially infinite imprecision errors, one coming from [a], the other from [b]. In the second case, you see that in fact, everything comes from [a] : the original imprecision error is magnified by the loop, in such a way that it becomes of the same order of magnitude as the real result just after 20 loops. The linear iterative scheme is unstable (just check the eigenvalues !).

```
main()
{
  float x,y,z;
  int i;
  x=1.0;
  z=1.0;
[a]  y=.618034;
  for (i=1;i<=20;i++) {
    z=x;
    x=y;
[b]    y=z-y; }
}
```

Now try this other program (EXAMPLES/gold2/gold2.c) which in theory should compute the same result as the first program, i.e. the gold number power 21. Just try with standard parameters : you will see two infinite imprecision errors for variable **t** coming from [a] and [b]. Now ask for unfoldloopnumber equal to 5 (or more) and let wideningthreshold be equal to 20, then you will see that the imprecision error is negligible (this is stable). In fact you can replace the number of iterations for the loop in the program by any number, even any variable (equal to $\top$) ; the analyser will prove that the error is always negligible.

```
main()
{
  float t,y;
  int i;
  t=1.0;
[a]  y=.618034;
```

```
  for (i=1;i<=20;i++) {
[b]    t=t*y; }
}
```

However in this case, the analyse still requires a certain amount of computations to get a finite estimation of the errors. And if we look more precisely at the influence of the **unfoldloopnumber** and **wideningthreshold** parameters, we notice that in all cases, a bounded estimation of the errors is obtained if and only if the product **unfoldloopnumber * wideningthreshold** is greater than a number which is around 95 to 105, depending on the value of **unfoldloopnumber**. This product represents the number of time iterations of the loop are computed before using widening, so that in all these cases, convergence is not due to the unfolding of loops, but to the fact that the computed growth of the errors induces no longer a growth of the error intervals (convergence due to the use of finite precision numbers for the analysis).

This rather disappointing behaviour when loops are unfolded, is in this example due to the fact that the value of y when entering the loop is a single value and not an interval, so that errors are computed exactly and their behaviour is chaotic, they do not necessarily decrease as would $ulp(y)$. If we substitute "y = __BUILTIN_DAED_FBETWEEN(0,.618034);" for "y=.618034;" in gold2.c, then the convergence for the errors is much more easily reached. If we unfold the loop at least twice (**unfoldloopnumber** greater or equal to 2), a bounded value for the error is got for any value of **wideningthreshold**. The bounds for the error obtained, when finite, for same **unfoldloopnumber**, may be a little smaller when y is a value and not an interval, but they are still comparable. And the computing time starting from an interval for y, and unfolding the loop at least twice, is on this example about 10 times smaller than starting from a value (whatever the value of **unfoldloopnumber**), because the fixpoint is got with only a few iterations.

The parameter **calculexacterr** can also be used to overcome this problem : setting it to 0 (errors computed exactly only outside loops) on example **gold2**, the analysis converges when unfolding at least 5 times, with any value of **wideningthreshold**.

We can also notice that first order errors converge whenever the loop is unfolded at least 3 times, and only higher order errors, which are in this case negligible, delay the convergence. Setting the parameter **relaxcvordresup** to 1, we then get the convergence unfolding only three times, but higher order errors are over-approximated.

Finally, executing 4 times (or more) the loop, with parameter **nbexecbeforejoin** set to 1, before beginning the unions over iterations, allows to obtain convergence without unfolding.

Now look at this other linear iterative scheme

EXAMPLES/pow_of_one_third7/pow_of_one_third7.c,

whatever the values of the parameters, you see that the initial error is not amplified (this scheme is stable for a wide class of input), and with **unfoldloopnumber** at least equal to 3, all errors are bounded.

```
#include <daed_builtins.h>

main(int n) {
  float x,y,z;
  int i;
  x=1.0;
  y = __BUILTIN_DAED_FLOAT_WITH_ERROR(-2.0,3.0,-0.01,0.01);
  y = y-1.0/3.0;
  for (i=1;i<=n;i++) {
    z=x;
    x=y;
    y=(x+z)/6.0; }
}
```

For the following unstable one (EXAMPLES/stability4/stability4.c), the analyser gives infinite bounds for the errors when we do not unfold completely the loop. These warnings are correct : see how the initial error can propagate – up to 10^{15}, the floating-point value may be completely meaningless ! – when the loop is completely unfolded, i.e., when unfoldloopnumber is set to a value greater or equal to $51 = 100/2 + 1$.

```
#include <daed_builtins.h>

main()
{
  float x,y,z;
  int i;
  x=1.0;
  z=1.0;
  y=__BUILTIN_DAED_FLOAT_WITH_ERROR(0,.618034,-0.00001,0.00001);
  for (i=1;i<=100;i++) {
    z=x;
    x=y;
    y=z-y; }
}
```

Now let us look again at linear iterative schemes, but this time for solving a simple set of linear equations by Jacobi's method :

$$\begin{cases} 2x + y &= \frac{5}{3} \\ x + 3y &= \frac{5}{2} \end{cases}$$

which solution is $x = \frac{1}{2}$ and $y = \frac{2}{3}$. The first method below uses the recurrence scheme :

$$\begin{cases} x_{n+1} &= \frac{5}{6} - \frac{1}{2} y_n \\ y_{n+1} &= \frac{5}{6} - \frac{1}{3} x_n \end{cases}$$

We want to see if, when we modify slightly the coefficients of this recurrence scheme, and when we have an uncertainty on the initial values, do we still get sensible results ?

This is program EXAMPLES/jacobi_stable/jacobi_stable.c :

```
#include <daed_builtins.h>

int main () {

  double x1,y1,x2,y2;
  double a,b,c,d;

int main () {

  double x1,y1,x2,y2;
  double a,b,c,d;
  int i;

  a = __BUILTIN_DAED_DOUBLE_WITH_ERROR(0.8,0.85,0.1,0.1);
  b = __BUILTIN_DAED_DOUBLE_WITH_ERROR(0.4,0.6,0.1,0.1);
  c = __BUILTIN_DAED_DOUBLE_WITH_ERROR(0.8,0.85,0.1,0.1);
  d = __BUILTIN_DAED_DOUBLE_WITH_ERROR(0.3,0.35,0.1,0.1);
  x2 = __BUILTIN_DAED_DOUBLE_WITH_ERROR(2.0,3.0,0.1,0.1);
  y2 = __BUILTIN_DAED_DOUBLE_WITH_ERROR(2.0,3.0,0.1,0.1);

  for(i=0;i<100000;i++) {
    x1=x2;
    y1=y2;
    x2 = a-b*y1;
    y2 = c-d*x1;
  };
}
```

This can be shown to be stable by the analyser. Either with **unfoldloopnumber** lower or equal to 20 and **unfoldloopnumber** * **wideningthreshold** greater than some number between 54 and 84 that depends on the value of **unfoldloopnumber**. Or with **unfoldloopnumber** greater or equal to 21 and any value of **wideningthreshold**. When **unfoldloopnumber** increases between two experiments, the bounds of the intervals usually get tighter.

Let us look now at Jacobi's method again, for a linear system

$$\left\{\begin{array}{rcl} 2x+3y & = & 3 \\ 4x+\frac{3}{2}y & = & 3 \end{array}\right.$$

which has the same solutions, but for which Jacobi's iterative method is now :

$$\left\{\begin{array}{rcl} x_{n+1} & = & \frac{3}{2}-\frac{3}{2}y_n \\ y_{n+1} & = & 2-\frac{8}{3}x_n \end{array}\right.$$

Same experiment as before : we perturb coefficients and initial data ; this is program

EXAMPLES/jacobi_instable/jacobi_instable.c :

```
#include <daed_builtins.h>

int main () {

  double x1,y1,x2,y2;
  double a,b,c,d;
  int i;

  a = __BUILTIN_DAED_DOUBLE_WITH_ERROR(1.2,1.6,0.1,0.1);
  b = __BUILTIN_DAED_DOUBLE_WITH_ERROR(1.2,1.6,0.1,0.1);
  c = __BUILTIN_DAED_DOUBLE_WITH_ERROR(2.0,2.5,0.1,0.1);
  d = __BUILTIN_DAED_DOUBLE_WITH_ERROR(2.5,3.0,0.1,0.1);

  x2 = __BUILTIN_DAED_DOUBLE_WITH_ERROR(2.0,3.0,0.1,0.1);
  y2 = __BUILTIN_DAED_DOUBLE_WITH_ERROR(2.0,3.0,0.1,0.1);

  for(i=0;i<100000;i++) {
    x1=x2;
    y1=y2;
    x2 = a-b*y1;
    y2 = c-d*x1;
  };
}
```

It is found to be unstable by the analyser, which it is indeed.

Now let us try to solve a system of differential equations by Euler's method. Try first the following program (see EXAMPLES/euler2/euler2.c) :

```
#include <daed_builtins.h>

int main () {

  double x1,y1,x2,y2;
  double h;
  int i;

  h = 0.1;
  x2 = __BUILTIN_DAED_DOUBLE_WITH_ERROR(0.99,1.01,0.0,0.1);
  y2 = __BUILTIN_DAED_DOUBLE_WITH_ERROR(1.99,2.01,0.0,0.1);

  for(i=0;i<100000;i++) {
    x1 = x2;
    y1 = y2;
    x2 = (1-2*h)*x1;
    y2 = (1+3*h)*y1 + h*x1;
  };
}
```

Notice that we have initial errors on **x2** and **y2**. The analyser is able to detect that this scheme is stable for **x2** and that it is unstable for **y2** (set **unfoldloopnumber** to at least 9).

3.2.2 Stability (higher-order)

The following example shows a stable non-linear scheme, EXAMPLES/x2_stable/x2_stable.c :

```
#include <daed_builtins.h>

int main (int n) {

  double x;
  int i;

  x = __BUILTIN_DAED_DOUBLE_WITH_ERROR(0.0,0.95,0.0,0.01);

  for(i=0;i<n;i++) {
    x = x*x;
  };
}
```

Stability is proven as soon as you choose unfoldloopnumber at least equal to 5.

Now the following similar non-linear scheme is stable at order one but not in higher order (take also unfoldloopnumber at least equal to 5). In fact, for values of **x** greater or equal to 1, this diverges indeed! This is program EXAMPLES/x2_instable/x2_instable.c :

```
#include <daed_builtins.h>
```

```
int main (int n) {

  double x;
  int i;

  x = __BUILTIN_DAED_DOUBLE_WITH_ERROR(0.0,0.95,0.0,0.1);

  for(i=0;i<n;i++) {
    x = x*x;
  };
}
```

3.2.3 Fixpoints of non-linear dynamical systems

The following example (EXAMPLES/KahanMuller/KahanMuller.c) is a classical example by W. Kahan and J.-M. Muller. If computed with exact real arithmetic, this would converge to 6. If computed with any approximation, this converges towards 100.

```
int main(void)
{
  float x0, x1, x2;
  int i;

  x0 = 11/2.0;
  x1 = 61/11.0;

  for (i=1 ; i<=13 ; i++)
  {
    x2 = 111 - (1130 - 3000/x0) / x1;
    x0 = x1;
    x1 = x2;
   }
}
```

Take unfoldloopnumber equal to 13, so that the loop is completely unfolded. Click on the button on the interface to see the estimated “real” value with respect to the estimated floating-point value, you will find approximations of 100 and of 6 respectively.

```
x0 : floating-point value [1.000000e2 1.000000e2]
     real value [5.84741865837139 5.97473790637248]
x1 : floating-point value [1.000000e2 1.000000e2]
     real value [4.67444955112023 7.03896201757234]
```

The estimation for the “real” value starts diverging. And it will be worse if we increase the number of iterations of the loop (and accordingly the unfoldloopnumber). Notice that the higher order error is the one that diverges.

We then can use the resource about the number of bits used in the analyser to encode real numbers (**mpfrbits**), in order to have more precise results for a larger number of iterations. The default number of bits is 60, set it to 65 for example : a much tighter estimate of the real result (and of the error committed) is obtained.

```
x0 : floating-point value [1.000000e2 1.000000e2]
     real value [5.91247078405563 5.91644923891061]
```

```
x1 : floating-point value [1.000000e2 1.000000e2]
     real value [5.88973283673352 5.96352951477832]
```

3.2.4 Progressive widening operators for errors

We have seen in section 3.2.1 how to use some parameters of the analysis (such as the number of initial executions or the number of unfolding in the loops) in order to get better fixpoint computation. We have also seen in section 3.2.3 how a higher number of bits for the numbers used for the analysis could be useful to have tighter approximations for some subtle (and unstable) algorithms.

On the contrary, it can also be useful to reduce the number of bits used for the approximation to improve the of convergence of fixpoint computations in loops. For example, take program **gold2** already studied in section 3.2.1. Suppose we do not make any execution of the loop before beginning the unions and we do not unfold the loop. Then the sequence of errors, if computed exactly, in real numbers, is ever increasing (to a finite limit). However, the computed sequence reaches a fixpoint after a finite number of precision, thanks to the use of finite precision numbers. Thus, with the default number of bits 60 for computing errors, 95 iterations are needed (we need the **wideningthreshold** parameter greater than 95 to get bounds for the error). Whereas with 30 bits of precision, only 50 iterations are needed to get the same result. Indeed, as the variables here are simple precision floating-point variables, 30 bits are highly enough to analyse the behaviour of errors.

On the basis of this idea, we propose widening operators for the computation of errors, more adequate to subtle numerical computations than jumping to fixed steps. On the same program **gold2**, we now keep the default precision of the analysis (**mpfrbits**) equal to 60, so that double precision computations in the program can be analysed with all the accuracy expected. But, setting the **progressivewidening** parameter to 1, we decide that this precision will be progressively degraded in loops, after the **widening threshold** is reached, and we still do not have a fixed point. Of course, it is mostly interesting to use this progressive widening with a very low value of **widening threshold**. Another threshold, **infthreshold**, sets the maximum number of iterations before widening towards infinity (classical widening). We also have two more parameters, **minprec** that determines the minimum precision under which we do not want to degrade the precision any longer, and **reductionstep**, that specifies of how many bits of mantissa we degrade the precision at each widening step.

Thus, setting for example **progressivewidening = 1**, **widening threshold = 2**, **infthreshold $\geq$ 32**, **minprec = 20** and **reductionstep = 2**, we find the same result as with the classical union + widening towards infinity, with only 37 iterations instead of 99 without progressive widening and **widening threshold = 95**.
Another advantage is that the progressive widening also uses the classical widening towards zero when values decrease towards zero. Thus, as soon as the computation of errors have converged, the fixpoint is got. And in the case of **gold2** for example, the number of iteration is independent of the value of **infthreshold**, provided that it is large enough. Whereas when we use the classical widening, we must iterate enough (95) for the errors to converge, but then the fixpoint is still not reached, it is reached only when the widening is used (for the value to converge towards zero). Thus, if the user chooses the **widening threshold** too small, the results are imprecise, and if too large, there are too many iterations. And finding the optimal value of **widening threshold** is not at all intuitive.

3.3 Integers

3.3.1 Conversion between integer types

Conversions between types are handled, with modular representation. Take
EXAMPLES/convunsintint/convunsintint.c :

```
int main() {
  unsigned int x;
  int y;
  x = 2*2147483647;
  y = (int) x;}
```

Variable x is 4294967294 with no error. When converted to a signed int, it exceeds the maximum possible value, thus the machine value is -2, with an error compared to a real computation (with infinite capacity of integers) equal to 4294967296. The result for the real computation is thus 4294967294, which is the expected value. We recall that the real result can be seen for example with the user interface, ticking the box close to the place where the interval value for the variable is displayed.

Let m and M be the minimum and maximum values that can take an integer of a given type. Because of the modular computation, we allow two possible abstractions for the set of values that can take an integer represented by a couple of integers (a, b) :

– the interval $[a, b]$ if $m \leq a \leq b \leq M$,

– or the union $[m, b] \cup [a, M]$ if $a > b$.

Indeed, when we have the operation $[a, M] + 1$, we want to be able to find the exact result $[a+1, M] \cup [m, m]$. With the interval representation, we would only be able to express that the result lies in the whole integer range $[m, M]$.

However, if an integer lying in a union $[m, b] \cup [a, M]$ has to be converted to a larger integer type or a floating-point type, the best estimate of the converted result is then $[m, M]$, because it is a tighter estimate than $[m', b] \cup [a, M']$ where m' and M' are the limits of the larger type.

Because of this representation, the behaviour in loops can be difficult to understand. For example, take **EXAMPLES/overint/overint.c** :

```
int main() {
  int x;
  x = 0;
  while (x>=0)
    x++;   }
```

The floating-point result found at the end of the loop is -2147483648, with a potential error in $[0, +\infty]$. This is indeed the best result that can be found because the loop is iterated until x reaches the maximum value for **int** type, then **x++** makes it loop to the minimum **int** value, -2147483648 : just after this instruction, we get $x = [-2147483648] \cup [1, 2147483647]$. Then the loop ends for $x = -2147483648$. With no limitation for integers, x would have gone to ∞ and the loop would have never ended, so there is a potentially infinite error.

Now, the resembling example where variable **x** is no longer of **int** type but **char** (see **overchar.c**) looks disappointing at first : x is found in $[-128, -1]$ when we expected to find exactly -128, still with an error in $[0, +\infty]$. Indeed, the loop test **x>=0** converts x in **int** type before making the comparison : when the maximum **char** value is reached, we have just before this test, $x = [-128] \cup [1, 127]$. Then the conversion to an **int** (larger type) gives $x = [-128, 127]$, and the negative values after the test are found to be $[-128, -1]$.

3.3.2 Particular case of the conversion float towards int

When the converted floating-point exceeds the capacity of the integer type of the result, the conversion is not achieved in the same way as between two integer types, with modular representation. Indeed, the result is not defined by the norm. In this case, the result is the whole range of the integer type, we deduce the error from that.

For example, take **EXAMPLES/cstefloatint/cstefloatint.c** :

```
int main() {
   float x;
   int y;
   x = __BUILTIN_DAED_FLOAT_WITH_ERROR(512,2147483649,0,4);
   y = (int) x;}
```

The analyser finds for the value of y the whole integer range. The minimum error is zero, the maximum error is 2147483649-**MIN_INT**+4.

Moreover, when a floating-point number is converted to an integer, all errors are agglomerated on one error point corresponding to the line of the conversion : indeed, if all errors are small, it would lose too much precision to overapproximate all errors from all sources by an integer, it is better to do it only once on the sum of errors. The drawback is that the source of existing errors is lost : if interested by the error due to a conversion, the user must look at the source of errors on the floating-point variable that was converted.

3.3.3 Widening operators for integer variables

The widening operator for integer variables had to be adapted to be used together with this modulo representation. Indeed, consider a limited loop on an integer counter, such as the following :

```
int main(void) {
  float y;
  for (int i=0 ; i<=100 ; i++)
   y = y+1;
}
```

With the classical widening to the maximum integer value after some number of iterations, at the iteration following the widening, the maximum integer value loops to the minimum, and an error is committed : we get after narrowing for i in the loop, $[m, 100]$ with a possibly infinite error (instead of $[0, 100]$ with no error). Whereas if using fixed steps in the widening, such as powers of 2 or 10, we use the information that i remains finite and does not loop to its minimum possible value m, and thus no error is committed.

3.3.4 Bitwise operations : use of subdivisions

It is possible to approximate quite accurately the result of a bitwise or shift operation on interval operands. But it is more problematic to propagate existing errors without overapproximation. The error coming from errors on the operands, is estimated by the difference between the result of the operation on the sums machine values plus errors, and the result of the same operation on machine values. This error is associated to the label denoting the bitwise operation (the source of existing errors is not tracked, the user has to look at the sources of errors on the operands). When the value is a point, this can be computed without losing too much precision. But when it is an interval, as we do not handle the dependency between the values, the error is overapproximated. For that, we propose a solution, that can be costly in computing time and memory space needed, but is efficient : we propose to subdivide value intervals for the computation of the errors. Reducing the width of the intervals reduces the explosion problem of the intervals. Even if the two operands have interval values, only one (the one with larger width) is subdivided. This possibility and the maximum number of subdivisions is a parameter of the analysis (used by setting the parameter **Interval subdivisions (bitwise operations)** in the **Misc** submenu that appears when the analysis is started) Take for example program **EXAMPLES/calculbet/calculbet.c** :

```
int main(void) {
  int i,j;
  i = __BUILTIN_DAED_INT_WITH_ERROR(1200129,1380006,-4,4);
  j = i & 0x7FFFF0F0;
}
```

With no subdivision of intervals, we get the value of j in $[1200128, 1376496]$, with an error in $[-180464, 176368]$ in 4 M of memory and 0.02 seconds of analysis time. If subdividing at most 500000 times for example, we get the same interval for values, with an error in $[-4096, 4096]$, in 14 M of memory space and 6.5 seconds of analysis time. In that case, all values in the interval are enumerated. This analysis can be costly for large programs, and we can try compromises : for example, with 50 subdivisions, we get an interval for the error just slightly larger, $[-4336, 4336]$, with 4 M of memory and only 0.02 seconds of analysis time, that is the same needs as for the analysis without subdivisions.

Bitwise operations have a clear meaning only when integers are coded on a fixed number of digits. Thus, when encountering a bitwise or a shift operation, we reduce the interval for the real value, got by the sum of the machine interval and the error, to the values belonging to the considered integer type. For example, if i is an operand of a bitwise operation, and is of an unsigned integer type, if $i = [0, 10] + [-1, 1]\varepsilon$, we take as interval for the real value $i = [0, 11]$.

3.4 Complex data structures

A basic alias analysis can be added, setting the **aliases** parameter to 1 or 2, (it should be set to 0 for an analysis without alias, more efficient if the program does really contain no complex data structure). For structs and pointers, there is no difference between the analysis with parameter 1 or 2, the difference arises for arrays. Here are some simple examples demonstrating the features of alias analysis.

3.4.1 Structs

Now we can define ANSI C **struct** as in the following example (**EXAMPLES/alias3/alias3.c**) :

```
typedef struct pair {
  float x1;
  float x2;
} twofloats;

void main() {
  twofloats X;
  X.x1=1.1111111111;
  X.x2=2.2222222222;
}
```

Names that are known to the analyser are “dereference” names : hence one would find in the variables’ list **X.x1** and **X.x2**. The different fields of a structure are always distinguished (unlike for arrays with **aliases** parameter set to 1).

3.4.2 Arrays

We can also consider arrays of values as in the following code (**EXAMPLES/alias1/alias1.c**) :

```
void main() {
  float a[2];
  a[0]=1.111111111111;
  a[1]=2.222222222222;
}
```

If the **aliases** parameter is set to 1, the analyser will only know of the array **a[]** as a whole ; i.e. the result in the variables’ list will be the only variable **a[]**, with floating-point value between **1.111111...** and **2.222222...**.

If it is set to 2, the analyser will distinguish all elements of the array : the analysis is more accurate, and of course more costly.

3.4.3 Pointers and aliases

Pointers can be considered (**EXAMPLES/alias2/alias2.c**) :

```
void main() {
  float x;
  float *y;
  float z;
  z=1.11111111111;
  y=&z;
  x=*y;
}
```

Variable **y** is not in the list of variables, it is known to be an alias by the analyser. The floating point values known to the analyser are that of **z** and **x**. The analyser finds out that $x = z = 1.11111....$

We can also consider also any complex data structures, such as here in example **alias4.c**, arrays of **struct** elements.

```
  struct st {
    float x1;
    float x2;
  };

main() {
  struct st tab[10];
  tab[0].x1=2.22;
  tab[0].x2=3.33;
}
```

Relevant entries in the list of variables would be **tab[].x1** and **tab[].x2**.

3.4.4 A more complex example

Here is now a code which is a typical finite differences scientific code (**EXAMPLES/chaleur_1D/chaleur_1D.c**). It is an implicit 1st order finite difference method for solving the one-dimensional heat equation, with initial values discretized on a 12 points line, equal to its coordinate. The boundary conditions are that the extrema have value zero : thus in the following code, we see that the heat decreases on each of the 12 points of the line. This scheme is stable.

Setting the **aliases** parameter to 1, the analyser sees the array as a whole, and the analyser cannot see that the scheme is stable.

Setting it to 2, and setting **unfoldloopnumber** and **wideningthreshold** to sufficiently high values, for instance to 150 and 50 respectively, the analyser finds the scheme stable (with errors at most of order 10^{-5}).

```
main()
{
  float ut[12];
  float vt[12];
  float alpha;
  int i, j;
  alpha = .2;

  for (i=1;i<11;i++)
    ut[i] = i;
  ut[0] = 0;
  ut[11] = 0;
  for (j=1;j<100;j++) {
    for (i=1;i<11;i++)
      vt[i] = (1-2*alpha)*ut[i]+alpha*(ut[i+1]+ut[i-1]);
    for (i=1;i<11;i++)
      ut[i] = vt[i];
  }
}
```

3.5 Some special uses of builtins

3.5.1 Helping the analyser

If you erase the **BUILTIN** constructs at lines [a] and [b] in the following code (EXAMPLES/E1/E1.c), this will result in a potentially infinite error for **t** coming from line [0] **y=y+x**. As a matter of fact, if the error on **t** was continuously equal to 0.001 and its floating point value would keep on being equal to 0, then the "real" result would be 10 (in fact, in other situations, it could go as far as a difference of 20), whereas the floating-point result would be zero ; the analyser extrapolates this imprecision error and pretends it may be infinite.

In the following code, at line [a] we specify that the error committed for variable **z** at this specific line cannot exceed the interval [-20.0,20.0]. Then we use this extra variable **z** to limit the error on variable **y** so that the error coming from line [a] is truncated for it does not exceed 20 in absolute value.

Notice the warnings at the console, indicating that the **if** statement in function **integre** may be unstable. This is program D1.c :

```
#include <daed_builtins.h>

float y,z,t;

void integre(float x) {
[0]  y = y+x;
  if (y>10.0)
    y = 10.0;
  if (y<-10.0)
    y = -10.0;
[a]  __BUILTIN_DAED_LIMIT_ERROR(y,z);
}

int main() {
  int i;
  y = 0.0;
[b]  t = __BUILTIN_DAED_FLOAT_WITH_ERROR(0,1,-0.001,0.001); \
     z = __BUILTIN_DAED_FERROR_MAX(-20.0,20.0);
  for (i=1;i<10000;i++)
    integre(t);
}
```

3.5.2 Creating stubs

In the following program (see EXAMPLES/E3/E3.c), we define two "stubs" for a special absolute value kind of function **new_abs** and for a home made square root function **home_made_sqrt**. Our absolute value function is abstracted by the sup of its argument with its argument plus one. For the square root function, we first assert that the argument should be positive, by **__BUILTIN_DAED_LIMIT(t,0,HUGE_VAL) ;**. Then the square root is approximated by floating-point interval [0,1] when the argument is less than 1, with error between -0.0000001 and 0.0000001. Finally, for an argument greater or equal to one, we use the fact that the square root is below its argument and is greater or equal to one. We add up an error of magnitude at most 0.0000001.

```
#include <daed_builtins.h>

float new_abs(float t) {
  float res;
  res = __BUILTIN_DAED_JOIN(t,-t+1);
  return res;
}

float home_made_sqrt(float t) {
  float res;
  __BUILTIN_DAED_LIMIT(t,0,1000);
  if (t < 1.0)
    res = __BUILTIN_DAED_FLOAT_WITH_ERROR(0.0,1.0,-0.0000001,0.0000001);
  else
    res = __BUILTIN_DAED_JOIN(__BUILTIN_DAED_FBETWEEN(1.0,1.0),
                              __BUILTIN_DAED_FEPSILON(-0.0000001,0.0000001)+t);
  return res;
}

main() {
  float x,y;
  x = __BUILTIN_DAED_FLOAT_WITH_ERROR(-100.0,100.0,-0.001,0.001);
  if (x > 10.0)
    x=15.0;
  if (x < -2)
    x=0;
  x = new_abs(x);
  y = home_made_sqrt(x);
}
```

3.5.3 Specifying constants

The following program (EXAMPLES/E4/E4.c) shows how to define constants in three different manners :

```
#include <daed_builtins.h>

main() {
  float x,y,z;
  x = .3333333;
  y = __BUILTIN_DAED_FBETWEEN(.3333333,.3333333);
  z = __BUILTIN_DAED_FLOAT_WITH_ERROR(.3333333,.3333333,.0000000001,.0000000001);
}
```

– constant **y** is considered exact (up to the precision of the analyser), meaning that the real number corresponding to it is exactly the floating point number representing the decimal string .3333333 (see the value of the variable **x**).
– constant **x** has an error associated to it since constants in C are in double precision and **x** is a **float** : this is the truncation error due to the implicit cast operation from **double** to **float**.
– constant **z** is declared with a specified error of .0000000001.

A current limitation of the analyser is that constants are read with double precision : they are considered exact in double precision even if they are not exactly represented by a double. This limitation can however be bypassed using the assertion

```
double x;
x = __BUILTIN_DAED_DBETWEEN_WITH_ULP(a,a);
```

that specifies that variable x has a floating-point value equal to the floating-point representation of a, and it may have an initial rounding error bounded by [-ulp(a),ulp(a)]/2. The drawback is that the error will be over-approximated, particularly in the case when a is exactly representable by a double.

3.6 Unstable tests

We call *unstable test* a test (for example '**if** ($x \leq 10$) {...}') where, due to the error on the variables involved, the branch taken in a computation achieved with floating-point numbers is not the same branch that would be taken in the same computation with real numbers.
All possibly unstable tests are indicated on the console during the analysis by warning messages specifying the number of the line in the source code where the test occurred. They are also indicated in a file **unstable_test.out**. But, to compute the errors committed with respect to the computation with real numbers, we make the assumption that the paths followed by the computation with real and floating-point numbers are the same. A discussion of this choice is given at the end of this section.

Two different messages are possible, depending on whether the test is sure to be unstable, or whether it may or not be unstable. Consider the following example
(see EXAMPLES/unstable_test1/unstable_test1.c) :

```
#include "daed_builtins.h"

main() {
  float x, y;
  x = __BUILTIN_DAED_FLOAT_WITH_ERROR(10,10,0.1,0.1);
  if (x <= 10)
    y = x + 1;
  else
    y = x - 1;
}
```

The test 'if (x $\leq$ 10)' is unstable. Indeed, for floating-point values, we have $x \leq 10$, the statement $y = x + 1$; is executed. But for the real value, $x = 10.1 > 10$, the statement $y = x - 1$; would be executed.

Now consider the program (EXAMPLES/unstable_test2/unstable_test2.c) :

```
#include "daed_builtins.h"

main() {
  float x, y;
  x = __BUILTIN_DAED_FLOAT_WITH_ERROR(10,10,-0.1,0.1);
  if (x <= 10)
    y = x + 1;
  else
    y = x - 1;
}
```

The test 'if (x $\leq$ 10)' is potentially unstable, the two branches are possible for the real value depending on the error : the test is unstable when the error on x is strictly positive, and stable when it is negative or zero.

On such examples as those shown above, the unstable test can be the origin of important differences between the computation in real numbers or in floating-point numbers (the computed value of y in unstable_test1.c is 11 and the value which is considered as the "real" value by the analyser is 11.1, whereas the real value is 9.1), and these differences are not taken into account by the current analysis.
On the contrary, when the outputs of a program containing some tests are continuous with respect to the inputs of the tests, which is often the case in practice, unstable tests are not such a threat. Consider for example

```
#include "daed_builtins.h"

main() {
  float x, y;
  x = __BUILTIN_DAED_FLOAT_WITH_ERROR(10,10,0.1,0.1);
  if (x <= 10)
    y = 2x;
  else
    y = 3x - 10;
}
```

The computed value of y is 20, the value considered as the real value by the analyser is 20.2, and the real value is 20.3. And the smoother the functional around the test, the smaller the error with respect to the real value.
It is thus important that the user looks at all unstable tests indicated, and tries to determine their consequence on the results that follow.

One reason why we make the assumption that the paths followed by the computation with real and floating-point numbers are the same, is the following : consider a test $x < y$ where the floating-point values of x and y are in intervals that have a non empty intersection. If one of the two variables presents an error, even very small, the analyser detects a possibly unstable test. But practically, if the error is very small, the real and floating-point paths will almost always be the same. Thus, taking in account all possible paths in the computation of errors may be a very pessimistic over-approximation, in addition to the fact that it is quite complex to achieve.

It may however be proposed as an option of the analyser in the future, to cover also the path taken by the real value, and when the paths meet again, to take the distance between the estimation of the real value in the two paths as an approximation of the additional error due to the unstable test. The 'main' path remains the floating-point path, and an over-approximation of the error committed by taking the wrong path is added to the classical errors.

3.7 Unspecified behaviours

The computation of the superset of possible floating-point values relies on the assumption that we know for sure what operations are executed. First we suppose the order of evaluation of arithmetic expressions (the more natural, left to right, innermost-outermost evaluation), and that the compiler makes no reorganisation. Second we assume that the intermediate computations in arithmetic expressions are rounded according to the IEEE format, and not in larger registers. Unfortunately, this is frequently not the case.

3.7.1 Evaluation order in arithmetic expressions

Consider on Pentium the following example (see EXAMPLES/evaluation_order1/evaluation_order1.c) :

```
int main()
{
  int i;
  float a,b,c,d;
  float x,y;

  a = 90.53;
  b = 15.32;
  c = 10.78;
  d = 95.07;

  x = 0;
  y = a + b - c - d;
  for (i = 1; i <= 10000 ; i++)
    x += i*y;
}
```

Let us first look at the result of the execution of this simple program with different compilation options :

```
gcc -[/g/O0] evaluation_order1.c
a.out : > x = -4.768648e+01
gcc -[O1/O2/O3] evaluation_order1.c
a.out : > x = 0
```

The different possibilities in the order of evaluation of the three arithmetic operations giving y lead to this dramatic difference on the value of x. This with or without the presence of parentheses to indicate the order of evaluation ; they are not taken into account. Indeed, these operations are associative for a computation in real numbers, but not in floating-point numbers, due to the rounding errors. If computed exactly, y and thus x are equal to zero. But, in simple precision floating-point numbers, rounding errors may be committed on the result of some of these addition and subtractions, and these errors depend on the values of the operands, thus on the evaluation order. This example is designed to deliberately amplify this rounding error, but this situation can very well occur in real case examples, and the user must be aware of the danger. Splitting the computations as

```
y = a+b;
y = y-c;
y = y-d;
```

and

```
y = a+b;
y = y-d;
y = y-c;
```

allows to understand what computations are really executed. However, if using the optimising options of the compilers, even splitting computations using intermediary variables is not enough to ensure what really happens.

Now we consider the analysis. The computation of the 'real' value should not really depend on evaluation order (but as the analysis uses also finite precision numbers, the tightness of the enclosure of the 'real' value depends itself on the evaluation order). But, to be able to compute the superset of the floating-point value, the analyser has to make an assumption on the evaluation order of the operations : we take for that the natural order, from left to right, and innermost-outermost. This way, the floating-point result is $x = 0$: we can note that it is not the same result as the one given by a compilation with no optimisation option !

Remark : the 'real' value computed for x by the analyser, when unfolding completely the loop for more precision, is $4.44134e-07$, whereas, if computed exactly, y and x are equal to zero. This is due to the fact that constants a, b, c and d are not represented exactly in double precision floating-point numbers, and these rounded values are the values considered as 'real' by the analyser (see for example section 3.5.3 for details on this current limitation of the analyser).

3.7.2 Storage of intermediary floating-point results in larger registers

Consider the following example.

```
int main () {
int tab[2]={0xffffffff,0x408c1fff};
double E1=*(double *)tab; /* 900.0 - ulp(900.0) */
        double E2 = E1+180.0;
int pe=(E1+180.0)/360.0;
int pe2=E2/360.0; }
```

The execution on a Pentium finds **pe=2** and **pe2=3**. Indeed, in the computation of **pe**, the result of **E1+180** is not rounded to the IEEE format (storage in a register), whereas we enforce this rounding by using an intermediary variable in the computation of **pe2**.

4 Use of the weakly-relational domain for the floating-point value

Many of the examples presented in the previous section have better results with this domain. We just give here a few examples showing the improvement over the non-relational domain.

4.1 Iterations of x=x-ax

Consider the following toy example, **EXAMPLES/loop_x-ax/loop_x-ax.c** :

```
int main(void)
{
  int i;
  double x;
  x = __BUILTIN_DAED_DBETWEEN(0,1);
  for (i=0 ; i<=10 ; i++)
    x = x - 0.2*x;
}
```

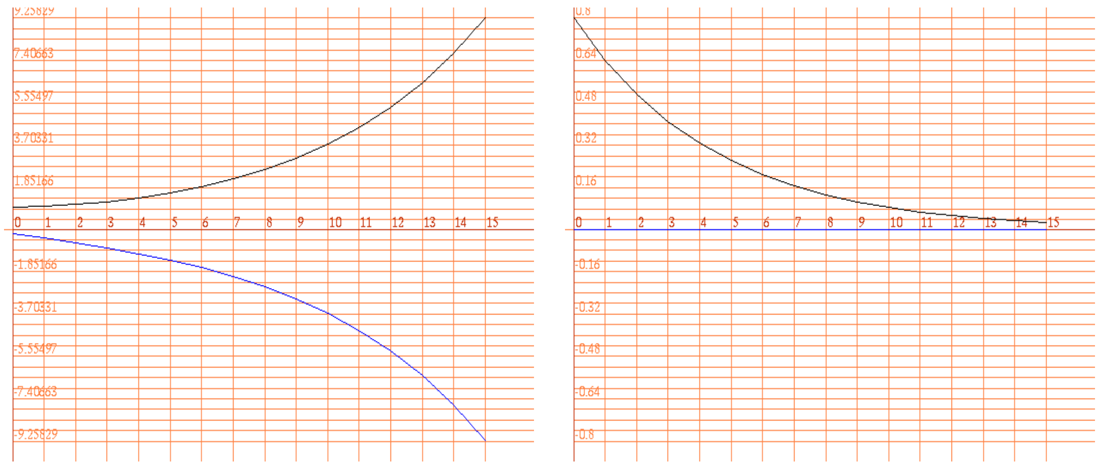
When the loop is unfolded (using **initial unfolding** or **unfoldloopnumber**), we observe that the value interval for x explodes with the non-relational analysis, whereas with the relational one, it decreases towards zero (see Figure 14). When the loop is not completely unfolded, we get $x = \top$ with the non relational analysis, whereas we get x bounded in $[0, 0.8]$ with the relational one.

4.2 A more general order 1 filter

The following example (**EXAMPLES/order1_filter/order1_filter.c**) is an extended version of the iterates of $x = x - ax$: in particular, an input with at each iteration a new value in the same interval is added, and the parameter a is no longer constant, but can take its value in an interval :

```
  double S0, S1, E1, a;
  int i;

  a = __BUILTIN_DAED_DOUBLE_WITH_ERROR(0.15,0.25,-0.00001,0.00001);
```



Non relational Relational

FIGURE 14 – Bounds on the floating-point value for 15 unfolded iterates

```
S0 = __BUILTIN_DAED_DOUBLE_WITH_ERROR(-10,10,-0.0001,0.0001);
S1 = __BUILTIN_DAED_DOUBLE_WITH_ERROR(-10,10,-0.0001,0.0001);

for (i=1; i<1000;i++)
{
  E1 = __BUILTIN_DAED_DOUBLE_WITH_ERROR(-10,10,-0.0001,0.0001);
  S1 = S1 - (2.0*S1 - S0 - E1) * a;
  S0 = E1;
}
```

The non relational analysis suffers from the same problem as for the loop $x = x - ax$: the expression

$$S1 - (2.0 * S1 - S0 - E1) * a$$

must be factorised as

$$S1 * (1 - 2.0 * a) + (S0 + E1) * a$$

for the analysis to converge. Indeed, the analyser can not recognise that the same value of $S1$ is used in different parts of the expression.

On the contrary, the weakly relational computation finds the same bounds for the output $S1$, with both versions of the expression : if parameter a is a constant fixed in $[0.15, 0.25]$, the output is found to remain in $[-10, 10]$, which is the real result. However, if a is defined in this interval by an assertion, then the output, if the widening threshold is large enough, is found to be bounded in $[-16.7, 16.7]$ (see Figure 15). These bounds are not optimal though the same result is found when rewriting the program. Indeed, our domain is designed to take care of affine correlations between variables, and here some relations are not affine when multiplying $E1$ and a is a is also in an interval. In this example, one way to improve the results would be to use the subdivisions of interval, subdividing for example interval a. However, this possibility is not yet available when several input intervals appear in a program, evne if only one is to be subdivided.

4.3 An order 2 filter

We now consider the following second-order filter, with constant coefficients but inputs in intervals :

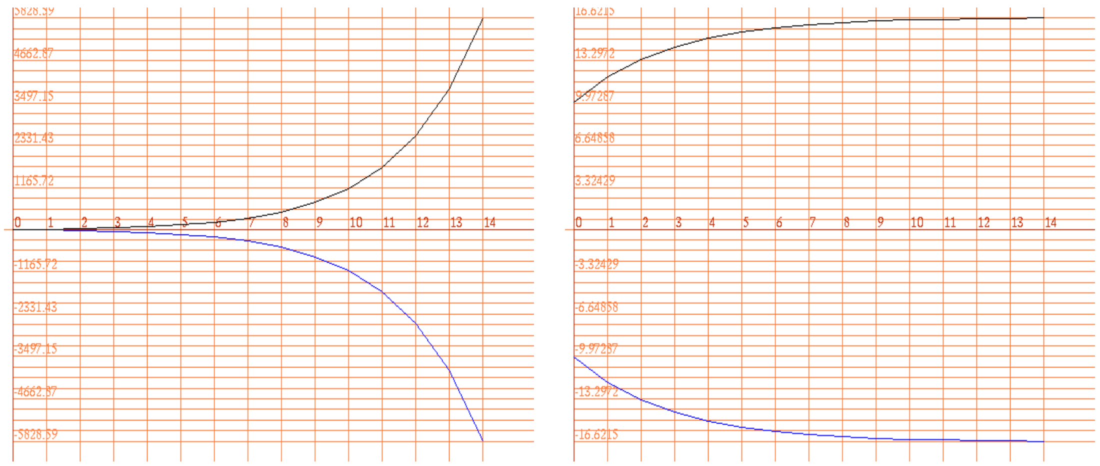


FIGURE 15 – Order 1 filter, bounds on the floating-point value for 15 unfolded iterates

```
double S,S0,S1,E,E0,E1;
int i;

S=0.0; S0=0.0; E0 = 0;
E=__BUILTIN_DAED_DBETWEEN(0,1.0);

for (i=0;i<=10;i++) {
  E1 = E0;
  E0 = E;
  E = __BUILTIN_DAED_DBETWEEN(0,1.0);
  S1 = S0;
  S0 = S;
  S = 0.7 * E - E0 * 1.3 + E1 * 1.1 + S0 * 1.4 - S1 * 0.7 ;
}
```

When unfolding the 11 iterates (with unfoldloopnumber or initial unfolding), but with depth of inter loop iterations relations equal to 1 (default value), we get comparable behaviours for the relational or non-relational analyses, even if the results are slightly more accurate with the relational analysis. This is due to interval arithmetic error explosion.
But increasing the depth of inter loop iterations relations with the relational domain dramatically improves the results, until keeping every relations, i.e. depth equal to 11 (see Figure 16).

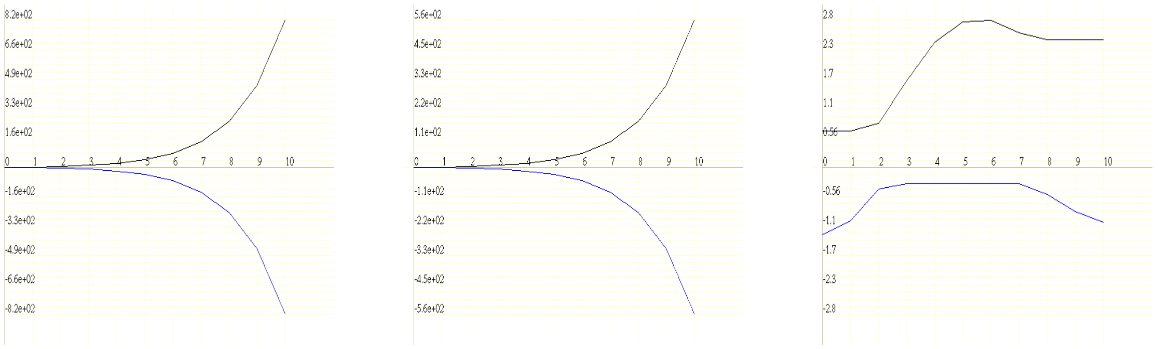
4.4 Use of assertions on the gradient in a loop

Now, we consider again the same filter of section 4.3, but we refine the information on input **E** by using

E = __BUILTIN_DAED_DGRADIENT(0.0,1.0,0.0,0.1,0.0,1.0,i,0)

in the loop, instead of **E = __BUILTIN_DAED_DBETWEEN(0,1.0)**. This specifies that E at iteration i, noted $E(i)$, is the intersection of the interval $[0,1]$ and of $E(0)+i*[0,0.1]$.

With the non relational domain, this brings no information compared to $E \in [0,1]$, and the results are same as before. but with the relational domain, the results are improved thanks to the weak relations between iterations (see figure 17). In particular, after a phase of initialisation, the analyser is able here to say that the output is always greater or equal to 0. However, we can also note that the upper bound on the output



a) non relational b) relational, no iterations relations c) relational, all relations

FIGURE 16 – Order 2 filter, bounds on the floating-point value for 11 unfolded iterates

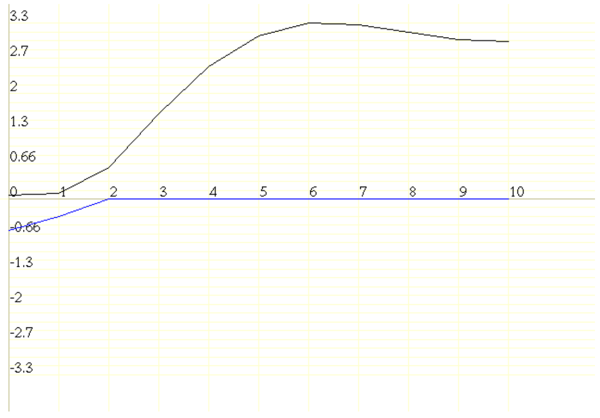


FIGURE 17 – Order 2 filter with an assertion on gradient.

is not always better than the upper bound obtained with an assertion only on the range of the variable : this is due to the fact that the intersection used on affine forms (and which is needed to intersect $[0,1]$ and $E(0)+i*[0,0.1]$) was not very carefully designed for the moment. We are working on this point.

4.5 Sensitivity analysis

We now show on two examples that compute by different ways the successive powers of the gold number, how the affine forms on the which the relational domain relies may be used for a sensitivity analysis to the inputs.

4.5.1 Printing format of an affine form

Let us first recall that information about the affine form representing a floating-point variable x can be printed using the assertion **x = __BUILTIN_DAED_FAFFPRINT(x,x)**, or its counterpart for double precision floating-point numbers. Each time the assertion is met by the analyser, information about the affine form is printed in the file **x.evolaffval** of the directory **Fluctuat_ProjectName/output**. This is an example of what is printed for one occurrence of the assertion :

```
interv flottant = 3.81964789e-1 3.81967261e-1
somme affine reelle = 3.81964789e-1 3.81967261e-1
erreur max = -6.20634100e-17 6.20634100e-17
somme affine flottant = 3.81964789e-1 3.81967261e-1
0 3.81966025e-1 3.81966026e-1
7 1.23606799e-6 1.23606800e-6
10 5.00000648e-13 5.00000649e-13
```

- on the first line is the bounds for the floating-point value of the variable, got by intersection after each operation of the interval got by interval analysis and the concretisation of the relational form,
- on the second line is the concretisation of the real affine form,
- on the third line is the maximum rounding error associated to the floating-point values,
- on the fourth line is the concretisation of the relational form (concretisation of the real affine form plus errors on bounds).
What really interests us, for the sensitivity analysis, is the coefficients of the affine forms, that are printed on the following lines. Each line begins by an integer i indicating a line in the source program (except that it is assumed to begin at line 2 !). Then the next two values on the same line indicate bounds for the corresponding coefficient, noted α_i^x in equation (4).

4.5.2 Unstable scheme (gold1.c)

Consider the following (unstable) computation of the gold number power 21. It is the body of the program **EXAMPLES/gold1/gold1.c** studied in section 3.2.1, in which we replace the input **y=.618034** by an assertion specifying that **y** is a double precision floating point number lying in the interval [.618033, .618035]. The successive iterates of this loop on **y=.618034** should converge towards zero, which is not the case, as we already saw. We show here that the instability of the scheme can be immediately observed on affine forms.

Variable **y** when entering the loop is represented as the affine form $\hat{y} = .618034 + e - 6\varepsilon_8$. This can be interpreted as an initial noise around the input value 0.618034.

```
double x,y,z;
int i;
x=1.0;
z=1.0;
y = __BUILTIN_DAED_DBETWEEN(.618033,.618035);   /* y=.618034;*/
for (i=1;i<=20;i++) {
  z=x;
```

```
    x=y;
    y=z-y;
    y = __BUILTIN_DAED_DAFFPRINT(y,y);
  }
```

Then at each iteration of the loop, the affine form for **y** is printed in the file **Fluctuat_ProjectName/output/y.evolaffval**. When completely unrolling the loop, we get :

```
interv flottant = 3.81965000e-1 3.81967000e-1
somme affine reelle = 3.81965000e-1 3.81967000e-1
erreur max = -2.77555756e-17 2.77555756e-17
somme affine flottant = 3.81965000e-1 3.81967000e-1
0 3.81965999e-1 3.81966000e-1
8 -1.00000000e-6 -9.99999999e-7
-
interv flottant = 2.36066000e-1 2.36070000e-1
somme affine reelle = 2.36066000e-1 2.36070000e-1
erreur max = -4.16333634e-17 4.16333634e-17
somme affine flottant = 2.36066000e-1 2.36070000e-1
0 2.36068000e-1 2.36068001e-1
8 1.99999999e-6 2.00000000e-6
-
interv flottant = 1.45895000e-1 1.45901000e-1
somme affine reelle = 1.45895000e-1 1.45901000e-1
erreur max = -8.32667268e-17 8.32667268e-17
somme affine flottant = 1.45895000e-1 1.45901000e-1
0 1.45897999e-1 1.45898000e-1
8 -3.00000000e-6 -2.99999999e-6
-
interv flottant = 9.01650000e-2 9.01750000e-2
somme affine reelle = 9.01650000e-2 9.01750000e-2
erreur max = -1.31838984e-16 1.31838984e-16
somme affine flottant = 9.01650000e-2 9.01750000e-2
0 9.01700000e-2 9.01700001e-2
8 4.99999999e-6 5.00000000e-6
-
interv flottant = 5.57200000e-2 5.57360000e-2
somme affine reelle = 5.57200000e-2 5.57360000e-2
erreur max = -2.18575158e-16 2.18575158e-16
somme affine flottant = 5.57200000e-2 5.57360000e-2
0 5.57279999e-2 5.57280000e-2
8 -8.00000000e-6 -7.99999999e-6
-
interv flottant = 3.44290000e-2 3.44550000e-2
somme affine reelle = 3.44290000e-2 3.44550000e-2
erreur max = -3.53883589e-16 3.53883589e-16
somme affine flottant = 3.44290000e-2 3.44550000e-2
0 3.44420000e-2 3.44420001e-2
8 1.29999999e-5 1.30000000e-5
-

[...]

-
interv flottant = -4.02700000e-3 4.33500000e-3
somme affine reelle = -4.02700000e-3 4.33500000e-3
erreur max = -1.14329119e-13 1.14329119e-13
```

```
somme affine flottant = -4.02700000e-3 4.33500000e-3
0 1.54000000e-4 1.54000001e-4
8 4.18099999e-3 4.18100000e-3
-
interv flottant = -6.77500000e-3 6.75500000e-3
somme affine reelle = -6.77500000e-3 6.75500000e-3
erreur max = -1.84988635e-13 1.84988635e-13
somme affine flottant = -6.77500000e-3 6.75500000e-3
0 -1.00000002e-5 -1.00000001e-5
8 -6.76500000e-3 -6.76499999e-3
-
interv flottant = -1.07820000e-2 1.11100000e-2
somme affine reelle = -1.07820000e-2 1.11100000e-2
erreur max = -2.99318621e-13 2.99318621e-13
somme affine flottant = -1.07820000e-2 1.11100000e-2
0 1.64000000e-4 1.64000001e-4
8 1.09459999e-2 1.09460000e-2
-
```

From this output file (in which we have suppressed some iterations for concision), we can already see in the first iterations that the computation scheme is unstable. Indeed, the set of real or floating-point numbers represented is at first decreasing in maximum amplitude, as it should be for the computation of the powers of the gold number. But the initial noise is immediately amplificated. We have, using for simplicity approximated scalar coefficients instead of the interval coefficients of the file :

$$
\begin{aligned}
\hat{y}_1 &= 3.82e^{-1} - e^{-6}\varepsilon_8 \\
\hat{y}_2 &= 2.36e^{-1} + 2e^{-6}\varepsilon_8 \\
\hat{y}_3 &= 1.46e^{-1} - 3e^{-6}\varepsilon_8 \\
\hat{y}_4 &= 9.02e^{-2} + 5e^{-6}\varepsilon_8 \\
&\ldots
\end{aligned}
$$

And indeed, after some time, the noise takes the advantage, for example we have for the last two iterates :

$$
\begin{aligned}
\hat{y}_{19} &= -1.00e^{-5} - 6.76e^{-3}\varepsilon_8 \subset [-6.77e^{-3}, 6.755e^{-3}] \\
\hat{y}_{20} &= 1.64e^{-4} + 1.09e^{-2}\varepsilon_8 \subset [-1.078e^{-2}, 1.11e^{-2}]
\end{aligned}
$$

Note also that no new noise symbols are created in the computations, because all operations are affine.

4.5.3 Stable scheme (gold2.c)

We now consider in the same way the stable computation of the gold number power 21 of program **EXAMPLES/gold2/gold2.c** :

```
double t,y;
int i;
t=1.0;
y = __BUILTIN_DAED_DBETWEEN(.618033,.618035);  /*  y=.618034;*/
for (i=1;i<=20;i++) {
  t=t*y; t = __BUILTIN_DAED_DAFFPRINT(t,t);
}
```

At each iteration of the loop, the affine form for **t** is printed in the file **Fluctuat_ProjectName/output/t.evolaffval**. When completely unrolling the loop, we get :

```
interv flottant = 6.18033000e-1 6.18035000e-1
somme affine reelle = 6.18033000e-1 6.18035000e-1
erreur max = -5.55111512e-17 5.55111512e-17
somme affine flottant = 6.18033000e-1 6.18035000e-1
0 6.18033999e-1 6.18034001e-1
7 9.99999999e-7 1.00000000e-6
-
interv flottant = 3.81964789e-1 3.81967261e-1
somme affine reelle = 3.81964789e-1 3.81967261e-1
erreur max = -6.20634100e-17 6.20634100e-17
somme affine flottant = 3.81964789e-1 3.81967261e-1
0 3.81966025e-1 3.81966026e-1
7 1.23606799e-6 1.23606800e-6
10 5.00000648e-13 5.00000649e-13
-
interv flottant = 2.36066844e-1 2.36069136e-1
somme affine reelle = 2.36066844e-1 2.36069136e-1
erreur max = -5.22351474e-17 5.22351474e-17
somme affine flottant = 2.36066844e-1 2.36069136e-1
0 2.36067990e-1 2.36067991e-1
7 1.14589807e-6 1.14589808e-6
10 6.18034703e-13 6.18034704e-13
110 3.09017400e-13 3.09017401e-13
-
interv flottant = 1.45897100e-1 1.45898989e-1
somme affine reelle = 1.45897100e-1 1.45898989e-1
erreur max = -4.61609371e-17 4.61609371e-17
somme affine flottant = 1.45897100e-1 1.45898989e-1
0 1.45898044e-1 1.45898045e-1
7 9.44271961e-7 9.44271962e-7
10 5.72950324e-13 5.72950325e-13
110 5.72949823e-13 5.72949824e-13
-
interv flottant = 9.01692225e-2 9.01706814e-2
somme affine reelle = 9.01692225e-2 9.01706814e-2
erreur max = -3.54679687e-17 3.54679687e-17
somme affine flottant = 9.01692225e-2 9.01706814e-2
0 9.01699519e-2 9.01699520e-2
7 7.29490221e-7 7.29490222e-7
10 4.72137303e-13 4.72137304e-13
110 7.08205293e-13 7.08205294e-13
-
interv flottant = 5.57275551e-2 5.57286371e-2
somme affine reelle = 5.57275551e-2 5.57286371e-2
erreur max = -2.53898930e-17 2.53898930e-17
somme affine flottant = 5.57275551e-2 5.57286371e-2
0 5.57280960e-2 5.57280961e-2
7 5.41019711e-7 5.41019712e-7
10 3.64746373e-13 3.64746374e-13
110 7.29491863e-13 7.29491864e-13
-

[...]

-
interv flottant = 1.73065288e-4 1.73075369e-4
```

```
somme affine reelle = 1.73065288e-4 1.73075369e-4
erreur max = -2.46336617e-19 2.46336617e-19
somme affine flottant = 1.73065288e-4 1.73075369e-4
0 1.73070328e-4 1.73070329e-4
7 5.04060603e-9 5.04060604e-9
10 3.85143403e-15 3.85143404e-15
110 3.08113256e-14 3.08113257e-14
-
interv flottant = 1.06960059e-4 1.06966636e-4
somme affine reelle = 1.06960059e-4 1.06966636e-4
erreur max = -1.59020915e-19 1.59020915e-19
somme affine flottant = 1.06960059e-4 1.06966636e-4
0 1.06963347e-4 1.06963348e-4
7 3.28833624e-9 3.28833625e-9
10 2.52033785e-15 2.52033786e-15
110 2.14227640e-14 2.14227641e-14
-
interv flottant = 6.61048462e-5 6.61091247e-5
somme affine reelle = 6.61048462e-5 6.61091247e-5
erreur max = -1.05056755e-19 1.05056755e-19
somme affine flottant = 6.61048462e-5 6.61091247e-5
0 6.61069854e-5 6.61069855e-5
7 2.13926694e-9 2.13926695e-9
10 1.64419217e-15 1.64419218e-15
110 1.47976511e-14 1.47976512e-14
-
```

On this example, at first iteration, we have $t_1 = .618034 + e^{-6}\varepsilon_7$. Then each multiplication adds a new symbol (at line 10), and at further iterations we agglomerate the older terms in a symbol corresponding to a fictitious line 110. We then observe that after the second iterate, the sum of the amplitude of these noise symbols begins to decrease, instead of being amplified as for the unstable scheme. This confirms that this scheme is stable.

4.6 Interpretation of tests with the latest fully relational domain

Consider the following program

```
#define epsilon 0.0000000001

void main() {
  double temp;
  int cond;

  temp = __BUILTIN_DAED_DREAL_WITH_ERROR(-2.0e-11,2.0e-11,-1.e-8,1.e-8);
  if (temp > epsilon)
    cond = 1;
  else if (temp < -epsilon)
    cond = 1;
  else
    cond = 0;
}
```

When the box **Tests interpretation on float only** in the **Loops** menu of the parameters window is not ticked, the tests are interpreted on floating-point and real values. This gives in the general case more accurate results then ticking the box in order to interpret tests (and narrow values) only on floating-point numbers, but may not give the expected results in some cases. Let us consider the analysis results of this example, with the box not ticked (and the latest fully relational analysis).

The real value for variable **temp** after the assertion is [-2.0e-11,2.0e-11], and considering the error, the floating-point value is in [-1.e-8,1.e-8]. After the analysis, variable **cond** has floating-point (int) and real value equal to 0 (no error), the real value of **temp** is in [-2.0e-11,2.0e-11] and the floating-point value is in [-1.e-10,1.e-10]. For the time being, errors are kept in [-1.e-8,1.e-8].

Indeed, the analyser when interpreting **temp** > **epsilon** and **temp** < **-epsilon** in real numbers, finds that this is never true. So, even if this may be true in floating-point numbers, it considers that, the control flows being the same, these branches are never taken and **cond** is always equal to 0. Moreover, he narrows the floating-point value of variable **temp**, according to the fact that **cond** is always equal to 0, the floating-point value of **temp** is found to be in [-1.e-10,1.e-10]. Supposing the value **cond** corresponds to the stopping criterion of an iterative algorithm, this would mean that we would have the number of iterations of the algorithm in real numbers, which may not be what is expected (still, the amplitude of the errors would indicate that the algorithm did not converge in floating-point values).

However, warnings are issued (that can be seen in the **View unstable tests** sub-menu of **Menu** in the main interface. They say (and same thing on line 11 for the second test)

```
Warning, a possibly unstable test occurred at line : 9 (real is bottom)
```

which is stronger that the usual warning that does not specify "real is bottom" : indeed, this indicates that a branch is not taken due to a test interpretation on the real value, that would have been taken if the test interpretation had been achieved on the floating-point value only. That allows the user to know that if he wants the actual floating-point control-flow, he should re-run the analysis with the box **Tests interpretation on float only** ticked. Then he gets, for the same example, that the real and floating-point value of **cond** are in [0,1], and that floating-point value of **temp** at the end of the program is in [-1.0e-8,1.0e-8].

Also, the same warning with "float is bottom" indicates that a branch is not taken due to a test interpretation on the float value, whereas the branch would be taken with real value (this is, we think, a less problematic case, as the user is more interested in general in the floating-point value flow).

4.7 Potentially local and irregular subdivisions

For the time being, the graphic interface of Fluctuat only allows the user to regularly subdivide some variables, for the whole program. But, using the language of assertions understood by fluctuat, the user can also himself (for the time being in the source code to be analysed) define some local and irregular subdivisions. For example, suppose he wants to compute the result of some function f for input E0 in $[1, 2^{21}[$ as in :

```
min = 1.0;
max = 2097152.0*(1-DBL_EPSILON); // 2^21*(1-DBL_EPSILON)
E0 = __BUILTIN_DAED_DBETWEEN(min,max);
S0 = f(E0);
```

But, suppose the computation of the result of this function uses the exponent of the input, and the user wants to avoid any unstable test around the changes of exponent. Then he can subdivide irregularly the input E0, in $[1, 2[, [2, 4[, [4, 8[, \ldots, [2^{20}, 2^21[$, call $f(E0)$ and then join the results for this partition of E0, by analysing instead the following program :

```
min = 1.0;
max = 2.0*(1-DBL_EPSILON);
E0 = __BUILTIN_DAED_DBETWEEN(min,max);
S0 = 1.0;
for (j=0; j<20; j++) {
  courant = f(E0)
  S0 = __BUILTIN_DAED_DJOIN(S0,courant);
```

```
    E0 = 2.0*E0;
  }
```

Note that this subdivision, on top of being potentially irregular, can also be only local to some part of the analysis (and thus less costly than the subdivision accessible via the interface) : indeed, after the values are joined (after the loop), the analysis can go on, without any subdivision. In the future, we will think about how to make these local subdivisions more transparent to the user.

When relations on the value thus computed are not necessary for further computations, the analysis will be faster if you use __BUILTIN_DAED_DCOLLECT instead of __BUILTIN_DAED_DJOIN (for non relational union of the results.

Finally, note that you still can use regular subdivisions via the graphic interface along with these manual subdivisions : for example here, the initial __BUILTIN_DAED_DBETWEEN(min,max) could thus be subdivided.

5 Implementation of the analyser

Our prototype is based on a simple abstract interpreter developed at CEA. For the time being, if it can parse all ANSI C, but most library functions like the trigonometric functions are not implemented. The main objection to their implementation is that there are no clear specification for these functions. The analysis is inter-procedural, using simple static partitioning techniques.

The interval representing the floating-point value is implemented using the precision of classical floating-point numbers, and higher precision is used for the errors (necessary because numerical computations are done on these errors, and they must be done more accurately than usual floating-point). We use a library for multi-precision arithmetic, MPFR, based on GNU MP, but which provides the exact features of IEEE754, in particular the rounding modes. The interface is based on Trolltech's QT and communicates with the analyser through data files.

5.1 Implemented operations

The following operations are interpreted by the analyser :

- arithmetic operations : $+$, $-$, $*$, $/$; division and modulo for integer operands
- the square root **sqrt** and the absolute value **fabs**
- cast operations
- prefix and postfix increment and decrement operations : $i{+}{+}$, $i{-}{-}$, ${+}{+}i$, ${-}{-}i$; compound assignment operations : $+=$, etc ...
- logical operations, AND (&&), OR (||)
- conditions in expressions, eg $x = (y \leq z)$
- tests : $<$, $\leq$, $==$, etc ...
- bitwise operations : AND (&), exclusive OR ($\wedge$), OR (|), complement (˜)
- bitwise shift operations : $<<$ and $>>$
- the **log** and **exp** operations are interpreted only for operands which floating-point value is a point (and not an interval). Indeed, the behaviour of these operations is not specified in the IEEE norm, and we can bound the error committed by using the floating-point operations only for a given input value. But perturbation errors of this input can be set, and a specification of bounds for values and errors for a specific implementation, if given, could easily be added.

5.2 Some specificities of the implementation

Integer computations (see section 3.3)

The modulo representation of integers is considered, and the user can specify on how many bits he wants every integer type to be represented. When a variable exceeds the maximum value of its type and loops to its minimum value (or the contrary), the difference between this value and the value that would be obtained if there was no limitation to integers, is taken as an error. Because of this modulo implementation, integer values are not always abstracted by an interval $[a, b]$, but also if necessary by a union $[\mathrm{MIN}, b] \cup [a, \mathrm{MAX}]$, where MIN and MAX are respectively the minimum and maximum possible values allowed by the type of the variable. Arithmetic operations and conversions are computed taking into account these two representations. This may sometimes lead to behaviours that are not obvious to understand, particularly in loops.

- bitwise operations : for these operations, we suppose that both the machine and the real value can be represented in the considered integer type. If necessary, when calling a bitwise operation, we reduce the errors on the operands so that the real value can not exceed the maximum integer of the type.
- conversion floating-point number $\rightarrow$ integer, division on integers, modulo and all bitwise operations : in order not to lose too much precision, the list of errors of the operand is transmitted on the result in only one error interval corresponding to the point of the operation (the provenance of errors is lost in these cases)

Floating-point computations

- unspecified behaviour (see section 3.7)
  - order of evaluation in arithmetic expressions : the computation of the superset of possible floating-point values relies on the assumption that we know the order of evaluation of arithmetic expressions (the more natural, left to right, evaluation), and that the compiler makes no reorganisation. Unfortunately, this is frequently not the case, and the compiler reorganises computations even in spite of parentheses. The most secure way to prevent this would be to split computations using intermediary variables, for example using an appropriate pre-treatment.
  - storage of intermediary results in internal registers, with a precision different from the IEEE floating-point : we have noted such behaviour, which results in that the values given by the analysis and the execution are different. Using new variables to assign intermediary computations in the source code should solve this problem.
- the **fabs** function applies to a double precision operand, and its result is a double precision number. Therefore, in examples when the operand is an interval not reduced to one value, even if we know that it is a simple precision number, a rounding error will be attributed to the computation.

Tests (see section 3.6)

Possibly unstable tests (when the machine and real values may not follow the same path in a test) are signalled by a message on the command line and in a file **unstable_test.out** in the **Fluctuat_ProjectName** directory, giving the line number in the source code where the test occurs. If there is no unstable test, this is also specified in the **unstable_test.out** file. For the time being, the errors are computed with the hypothesis that the execution of the program with real number would follow the same path as the execution with finite precision numbers. This can lead to false estimations of some errors.

Execution errors

We suppose that execution errors cannot occur. In case the analyser detects a possible error, it may signal it by a message on the command line but goes on with the analysis.

5.3 Iteration algorithm

When encountering a loop, which body is represented by a function f, the analyser computes an upper approximation of its fixpoint $\mathrm{lfp}(f) = \cup_n f^n(\perp)$ by an iterative algorithm, that solves the equivalent problem of finding an upper approximation for $\lim_n X_n$, where $X_0 = \perp$ and $X_n = X_{n-1} \cup f(X_{n-1})$.

Standard algorithm (with **unfoldloopnumber** = 1) : each time the analyser enters the body of the loop, it compares the present values of the variables with those got at the previous iteration :

  - if they are not comparable or if the new values are larger than the older, it takes as new values the union of the old and new (after a number of unions defined by the parameter named **widening-threshold**, a "widening" towards 0 or infinity is made so that we are sure this process ends within a reasonable number of iterations). And the remaining of the body of the loop is traversed with the new values
  - if the values of all variables are the same for the present and past iterations of the algorithm, a fix-point is reached, we quit the loop.
  - if the new values are included in the older ones, the old values give an approximation of a fix-point, but we try to get a better approximation by "narrowing" the old values using the new ones. The remaining of the body of the loop is traversed, then the analyser quits the loop (only one narrowing).

With **unfoldloopnumber** ≥ 2 :

In many cases, we noticed that with the previous algorithm, we get infinite bounds for variables whereas the program is stable, and there exists some simple finite bounds. This is due to the fact that when at each iteration we union the new values with the older one, we cannot see that for example the value of a variable decreases with the iterations, and that so does the error. Let us consider the following example :

```
t = 1;
for (i=1 ; i<=100 ; i++)
  t = t*0.618;             (epsilon)
```

With the above algorithm, we get :
$i = 1$:

$$t_1 = [0.618, 0.618] + 0.618\, ulp(1)\, [-1, 1]\epsilon$$

$i = 2$:

$$t_2 = t_1 * 0.618 = [0.618^2, 0.618^2] + 2 * 0.618^2\, ulp(1)[-1, 1]\, \epsilon$$

Noticing that $2 * 0.618^2$ is greater than 0.618, the union with the previous iteration writes

$$t_2 := t_2 \cup t_1 = [0.618^2, 0.618] + 2 * 0.618^2\, ulp(1)[-1, 1].$$

Indeed, we can only bound the error committed at each iteration by the ulp of the largest possible value for t, and this error adds to the previous error. As the maximum value for t, with the unions, is constant, the total error increases.
$i = 3$:

$$t_3 = t_2 * 0.618 = [0.618^3, 0.618^2] + (2 * 0.618^3 + 0.618^2)\, ulp(1)[-1, 1]\, \epsilon$$
$$t_3 := t_3 \cup t_2 = [0.618^3, 0.618] + (2 * 0.618^3 + 0.618^2)\, ulp(1)[-1, 1]\, \epsilon$$

The computed error is again increasing for the same reason ($2 * 0.618^3 > 0.618^2$), thus we finally get infinite bounds for the error ($]-\infty, +\infty[$), whereas the error really committed decreases, because the value of t decreases. In this particular case, we could get finite bounds with a (very) good widening, since the limit of the error given by this algorithm is $\frac{1}{1-.618} ulp(1)$. But there is a better solution, by unfolding twice the loop : let us rewrite the program as

```
t = 1;
for (i=1 ; i<=100 ; i++) {
  t = t*0.618;             (epsilon 1)
  t = t*0.618;             (epsilon 2)
}
```

With the same algorithm applied to this rewritten program, we get :
$i = 1$:

$$\begin{cases} t_1 = [0.618, 0.618] + 0.618\, ulp(1)\, [-1, 1]\epsilon_1 \\ t_2 = t_1 * 0.618 = [0.618^2, 0.618^2] + 0.618^2\, ulp(1)\, [-1, 1]\epsilon_1 + 0.618^2\, ulp(1)\, [-1, 1]\epsilon_2 \end{cases}$$

We can run two iterations before joining the values, thus the errors are smaller.
$i = 2:$

$$\left\{\begin{array}{l} t_3 = t_2 * 0.618 = [0.618^3, 0.618^3] + 2 * 0.618^3\, ulp(1)\,[-1,1]\epsilon_1 + 0.618^3\, ulp(1)\,[-1,1]\epsilon_2 \\ t_4 = t_3 * 0.618 = [0.618^4, 0.618^4] + 2 * 0.618^4\, ulp(1)[-1,1]\epsilon_1 + 2 * 0.618^4\, ulp(1)\,[-1,1]\epsilon_2 \end{array}\right.$$

$$\left\{\begin{array}{l} t_3 := t_3 \cup t_1 = [0.618^3, 0.618] + 0.618\, ulp(1)\,[-1,1]\epsilon_1 + 0.618^3\, ulp(1)\,[-1,1]\epsilon_2 \\ t_4 = t_4 \cup t_2 = [0.618^4, 0.618^2] + 0.618^2\, ulp(1)\,[-1,1]\epsilon_1 + 0.618^2\, ulp(1)\,[-1,1]\epsilon_2 \end{array}\right.$$

$i = 3:$

$$\left\{\begin{array}{l} t_5 = t_4 * 0.618 = [0.618^5, 0.618^3] + (0.618^2 + 0.618^3)\, ulp(1)\,[-1,1]\epsilon_1 + 0.618^4\, ulp(1)\,[-1,1]\epsilon_2 \\ t_6 = t_5 * 0.618 = [0.618^6, 0.618^4] + (0.618^3 + 0.618^4)\, ulp(1)\,[-1,1]\epsilon_1 + (0.618^5 + 0.618^4)\, ulp(1)\,[-1,1]\epsilon_2 \end{array}\right.$$

$$\left\{\begin{array}{l} t_5 := t_5 \cup t_3 = [0.618^3, 0.618] + 0.618\, ulp(1)\,[-1,1]\epsilon_1 + 0.618^3\, ulp(1)\,[-1,1]\epsilon_2 \\ t_4 = t_5 \cup t_3 = [0.618^4, 0.618^2] + 0.618^2\, ulp(1)\,[-1,1]\epsilon_1 + 0.618^2\, ulp(1)\,[-1,1]\epsilon_2 \end{array}\right.$$

Convergence is reached for the errors. We get from $t_4 \cup t_5$ that the error on t in the loop is in $(0.618 + 0.618^3)\, ulp(1)\,[-1,1] \approx 0.854\, ulp(1)\,[-1,1]$. We still have to iterate on the values to get the result that the floating-point value of t is in $[0, 0.618]$. We intend to ameliorate the widening operations to take into account such phenomena, in a future version of the tool : it is always more interesting to widen integer values first, then the floating-point range of the abstract values for floating-point variables, and then at the end, the imprecision errors.
Notice that we can also use only one same ϵ for the two unfolded lines (and this is what is practically implemented in the analyser). In some cases, it could lead to a quicker convergence for the errors.

This idea explains the existence of the **unfoldloopnumber** parameter : the user can decide whether he wants loops to be unfolded in the analysis, and how many unfoldings are wanted. In the above example, two unfoldings of the loop are enough to get good results, but in some other cases, more unfoldings will be necessary.
Suppose we unfold the loops N times and f is the functional associated with the body of the loop. The union (or widening) operator is applied in a cyclic way. That is, we compute an upper approximation to $\lim_n(X_1^n, X_2^n, \ldots, X_N^n)$ with X_i^n such that

$$\left\{\begin{array}{l} X_i^0 = \perp, \;\; i = 1 \ldots N, \\ X_i^n = X_i^{n-1} \cup f^{(i)}(X_N^{n-1}), \;\; i = 1 \ldots N,\; n \geq 1. \end{array}\right.$$

When the fix-point is found, the analysis goes on after the loop with $X = \bigcup_{i=1}^{N} X_i$.

This unfolding technique can also be used to unfold completely some loops. For instance when the number of iterations is not too big, and we either want a very tight approximation of the values after a stable loop, or the loop is unstable but we do not want to get infinite bounds.

5.4 Known limitations and bugs

5.4.1 Loops with composed entering/exit condition

The analysis on some examples with a loop which entering/exit condition is a composed condition, may not terminate. This happens in some cases when this condition includes the operator ||, for example in (see EXAMPLES/boucle_doubleconditions_2/boucle_doubleconditions_2.c)

```
int i = 0;
while ((i<=10) || (i<=10000))
  i++;
```

A solution in some cases is to reverse the order of the two conditions, writing here

```
int i = 0;
while ((i<=10000) || (i<=10))
  i++;
```

But most of the time, these forms are a problem for the analyser.

5.4.2 Structs and arrays

The arrays can be used as parameters of a function, but only as explicit pointers. For example, the following prototype of function *f*

```
voif f(int Tab[])
```

has to be rewritten as

```
void f(int *Tab)
```

A Installation of the tool

You can unzip the distribution by

```
unzip FLUCTUAT.zip
```

The distribution contains directories, under the main one called FLUCTUAT :
- **ABSINT**, contains directories with includes necessary to the analyser,
- **bin** contains all the binaries, in particular **daed_project** and **daed_analyse** which can be used for batch mode,
- **Config**, **LIB** and **SMP** contain a number of includes, libraries and config files for the analyser,
- **DEMO** contains a set of examples (some of which are explained in this manual),
- **MANUAL** contains the electronic version of this manual.

The binaries have been compiled, but some dependancies are needed :
- **libqt-mt** Qt graphical framework v3 libraries
- **libgmp** GNU Multi-precision library
- **libmpfr** Multiple-Precision Floating-point computations with correct Rounding library
- **gcc** GNU C Compiler

You also have to set **FLUCTUATHOME** environment variable to the root directory **.../FLUCTUAT** of the location at which you have installed the distribution.
We recommend to set your **PATH** variable with the following command :

```
 export PATH=$FLUCTUATHOME/bin:$PATH
```

To use the tool you now have only to type **fluctuat &**.

B Use of the tool in batch mode

Starting from the source file **example.c** :
- create the project file : **daed_project example.c example_.gph**
- set the parameters of the analysis :

  - create a resource file (for example **example.rc** given below) :
    - `* callstackdepth = 2 ;`
    - `* entryname = main ;`
    - `* narrowingthreshold = 2 ;`
    - `* wideningthreshold = 20 ;`
    - `* skiptemporaryvar = 1 ;`
    - `* lightcontext = 0 ;`
    - `* partitionlevel = 1 ;`
    - `* unfoldloopnumber = 1 ;`
    - `* initvar = 0 ;`
    - `* mpfrbits = 60 ;`
    - `* ...`
  - specify that this is the resource file to be used : **setenv TWORESOURCEFILE example.rc**
- start the analysis : **two_analyser -l -v example_.gph**

After analysis, there is in the directory named **output** :
- a file **liste_variables** containing the variables known at the end of the program.
- for each variable **xx** in **liste_variables**, a file **xx.var** with :
  - an interval including the floating-point value, printed with the precision of the **printf** function.
  - an interval including the real value (floating-point value + error)
  - an interval including the sum of errors
  - an interval including the sum of errors of order more than 2
  - for each line at which an error may have been committed that can affect **xx**, the number of the line, the names of the file and function including it, and the interval of this error after propagation through the program

Example of a result file **output/x.var** :

```
2.999982 2.999982                          // floating-point value
2.99998200000000 3.00298198200000          // real value
-1.18606567234280e-7 2.99986339343277e-3   // sum of errors
0 0                                        // error of order more than 2
147 test6_.gph main -1.18606567234280e-7 -1.18606567230811e-7
547 test6_.gph main 0 2.99998200000000e-3
```

Note that all parameters and options of the analysis accessible through the user interface (see section 2.1) can be set in the resource file, using keywords. Note also that not all of these parameters have to be set to a value, there are default values for all of them. Also, the resource files created by the user interace (with '.rc' extensions) can be edited and used for further analyzes, but these must be done in batch mode (using the interface erases the old resource file). Each line begins with a '*', then a keyword and its value preceded by '='. We describe here, for each option in the user interface, the corresponding keyword and if necessary its use :
- **Abstract semantics → analysiskind** 0 for non-relational analysis, 1 for weakly-relational analysis, and 2 for fixed-point analysis
- **Entry Name → entryname**
- **MPFR Bits → mpfrbits**
- **Maximum Iterations → maxiter**
- **Initialise variables to 0 → initvar**
  0 for an initialisation to 0, 1 to top
- **Symbolic Execution → execution**
  1 for a symbolic execution, else 0

- **Automatic Parameters** → **iterauto**
  1 for automatic parameters, else 0
- **stockage** → **stockage**
  1 for storage of all states, else 0 for partial storage
- **Do not limit call stack** → **nolimitstack**
  0 to limit the call stack (by the call stack depth parameter), else 1
- **Call Stack Depth** → **callstackdepth**
- sub-menu **Pointers** → **alias**
  Ticking the box **No pointers** corresponds to a value of **alias** equal to 0, **Abstract arrays as one location** to 1, and **Unravel arrays** to 2.
- **Initial Unfolding** → **nbexecbeforejoin**
- **Unfold Loop Number** → **unfoldloopnumber**
- **Depth of inter-loop relations** → **nbrelationlevel**
- **Widening Threshold** → **wideningthreshold**
- **Narrowing Threshold** → **narrowingthreshold**
- **Use progressive widening** → **progressivewidening**
  Ticking the box corresponds to **progressivewidening=1**, otherwise it is 0
- **Loss of precision (bits)** → **reductionstep**
- **Minimal precision** → **minprec**
- **Threshold before standard widening** → **infthreshold**
- **Exact error calculation for constants** → **calculexacterr**
  1 to always compute them the more accurately possible (equivalent to ticking the box in the user interface), else 0
- **Convergence of higher order errors** → **relaxcvordresup**
  1 to consider the convergence only of first order errors and total errors for fixpoint computation, else 0 to examine also the convergence of higher order errors (equivalent to ticking the box in the user interface)
- sub-menu **CrossPlatform** → one can specify **MIN_CHAR**, **MAX_SCHAR**, **MIN_SCHAR**, **MAX_UCHAR**, **MAX_SHRT**, **MIN_SHRT**, **MAX_USHRT**, **MAX_INT**, **MIN_INT**, and **MAX_UINT**.
- **Interval subdivisions (bitwise operations)** → **nbdivisions**
- **Line of variable entry** → **linenumber**
- **Number of subdivisions** → **slices**